Artificial Intelligence
CS 6364

Professor Dan Moldovan

Section 2

Problem Solving and Search

Solving Problems by Searching

Problem-solving methods refer to a large number of ideas that deal with deductions, inference, planning, common sense reasoning, theorem proving, etc.

The main tool for problem solving is search.

Problem-solving systems have 3 components:

1. Database which describes:
   - Goal to be achieved—goal state
   - Current situation—called state

   The database consists of current situation which changes constantly and goal state.
2. Set of operators that manipulate the database
3. Control strategy—decides what operator to apply and where to apply.

Applications of Problem-Solving

Problem-solving methods are used for:

- Finding solutions to puzzles
- Finding proofs for theorems in logic or mathematics
- Finding shortest path in a graph
- Game playing—decides sequence of moves
- Finds solutions to calculus problems (there is a system called SAINT—symbolic automatic integration)
- Others

Design of Agents for Problem-Solving

The design of agents for problem-solving methods may be broken down into:

a) Problem formulation

This is the process of transforming the problem to be solved into

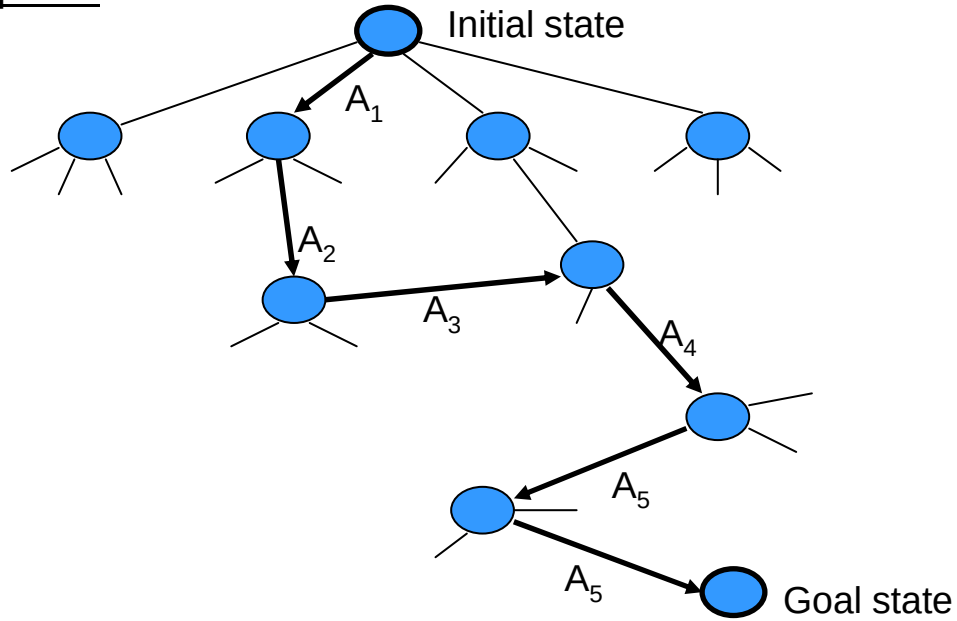
- Goal state
- Set of possible states that completely describe all situations
- Actions to be taken, or transitions from a state to another.

b) Search algorithm that takes the system from an initial state and provides a sequence of actions (and states) leading to goal. The solution is this sequence of actions.

Design of Agents for Problem-Solving

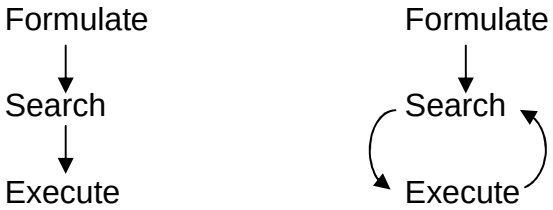
c) Execution phase is the implementation of actions.

Search space



A_i—actions—that cause transitions from state to state.

Several possibilities of problem-solving systems:



Example 1 (Water jug puzzle)

There are two empty jugs, one of 4 gallons, one of 3 gallons. Fill the 4-gallon jug with 2 gallons of water.

(a) Problem formulation

Decide how to represent states

s_o: (initial state) (0,0)

s_i: (state i) (x_i,y_i)

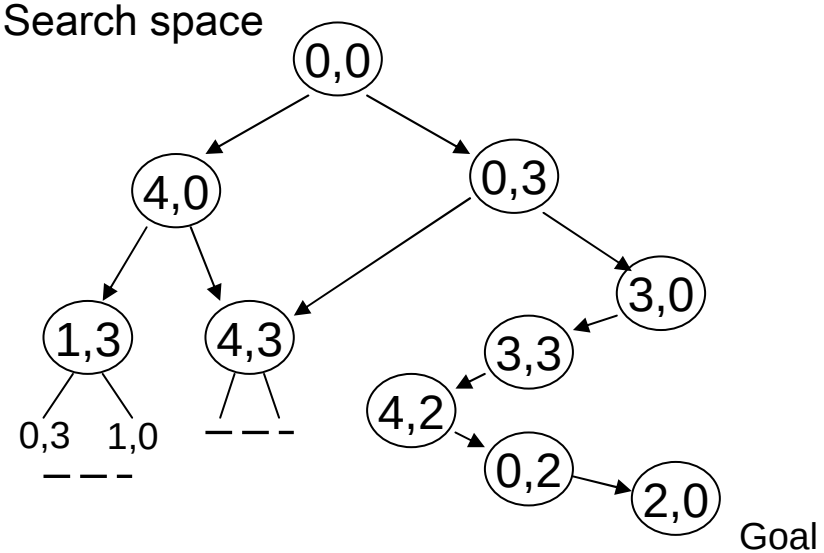
where: x_i—content of 4-gallon jug

y_i—content of 3-gallon jug

s_G: (goal state) (2,0)

Actions are to fill or empty the jugs

(b) Search space



Solution Path:

0 0

0 3

3 0

3 3

4 2

0 2

2 0

Production Rules for the Water Jug Puzzle

1	(x, y) if x < 4	→ (4, y)	Fill the 4-gallon jug
2	(x, y) if y < 3	→ (x, 3)	Fill the 3-gallon jug
3	(x, y) if x > 0	→ (x-d, y)	Pour some water out of the 4-gallon jug
4	(x, y) if y > 0	→ (x, y-d)	Pour some water out of the 3-gallon jug
5	(x, y) if x > 0	→ (0, y)	Empty the 4-gallon jug on the ground
6	(x, y) if y > 0	→ (x, 0)	Empty the 3-gallon jug on the ground
7	(x, y) if x + y ≥ 4 and y > 0	→ (4, y-(4-x))	Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full

Production Rules: Continuation

8	(x, y) if x + y ≥ 3 and x > 0	→ (x - (3 - y), 3)	Pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full
9	(x, y) if x + y ≤ 4 and y > 0	→ (x + y, 0)	Pour all the water from the 3-gallon jug into the 4-gallon jug
10	(x, y) if x + y ≤ 3 and x > 0	→ (0, x + y)	Pour all the water from 4-gallon jug into the 3-gallon jug
11	(0, 2)	→ (2, 0)	Pour the 2 gallons from the 3-gallon jug into the 4-gallon jug
12	(x, 2)	→ (0, 2)	Empty the 4-gallon jug on the ground

Example 2 (More complex Water jug puzzle)

There are three empty jugs, one of 100 gallons, one of 4 gallons and one of 3 gallons. You are to design an intelligent agent to fill the 100-gallon jug with 5 gallons of water.

Specifically show

a) Formulation of the problem by deciding how to represent an arbitrary state, the initial state and the goal state.

An arbitrary state is represented as (X, Y, Z) where X,Y,Z are the volumes of water in 100 gallon jug, 4 gallon jug and 3 gallon jug respectively.

- Initial state (0,0,0)
- Goal state (5,X,X) here x is don't care.

Example 2: continuation

b) Select a set of actions in the form of if-then rules that are used in your solution. Write the rules in plain English.

1) Rule fill jug if empty
   if jug X is empty, fill jug X to the rim.
2) Transfer water from jug to jug
   if jug X is empty and jug Y has water, then move water from jug Y to jug X

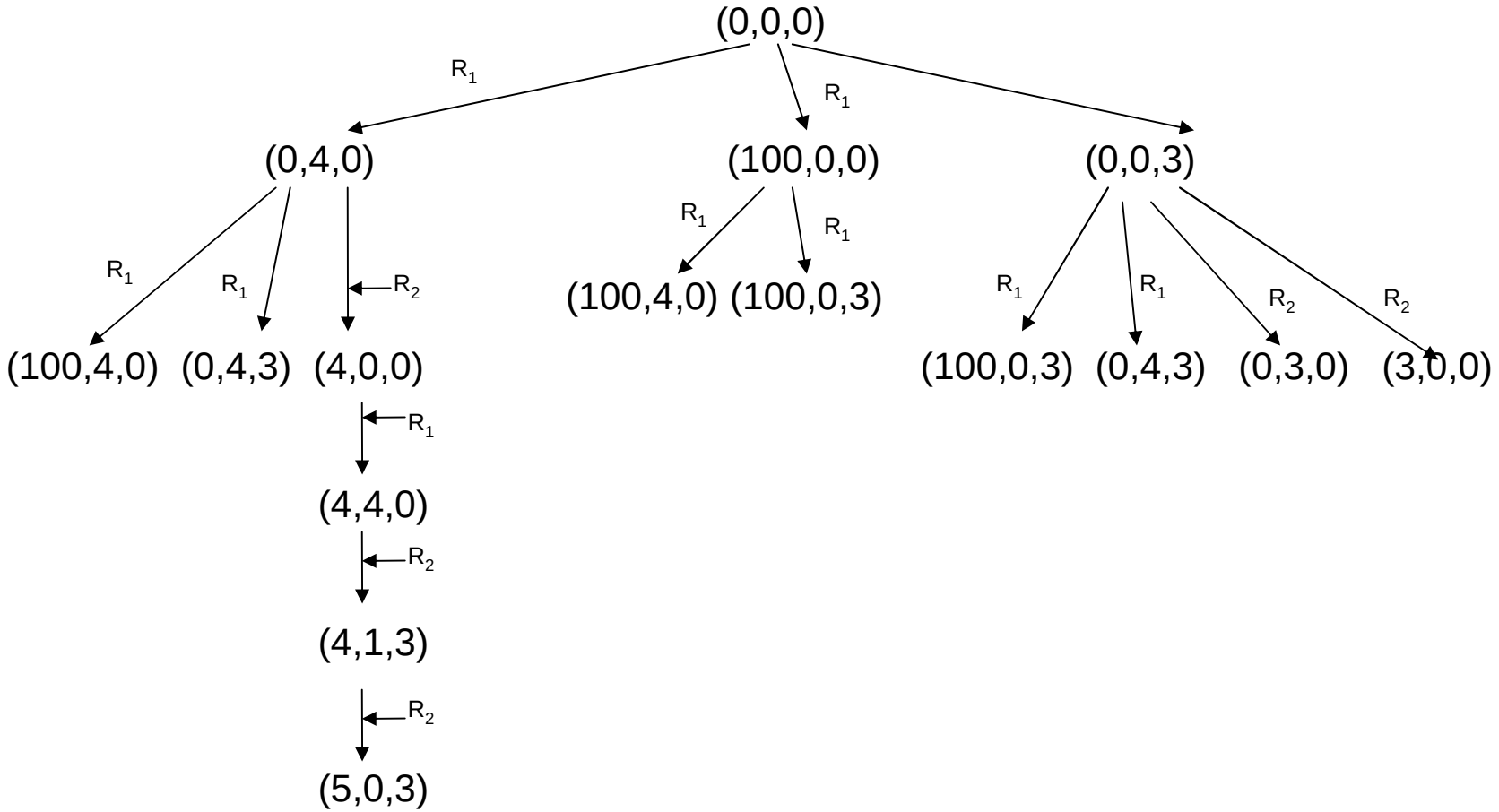
   (condition X > Y)

Note
1. Other possible solutions
   a) 5= (3+3+3)-4
   b) 5= (4+4)-3
   c) 5= (4-3)+4

2. The solution above is
   5=4+(4-3)

Example 2: continuation

c) Show the search space of your design and the path to the solution. When showing the search space the first 2 levels are sufficient, but you need to show the complete path to the goal.



Example 2: continuation

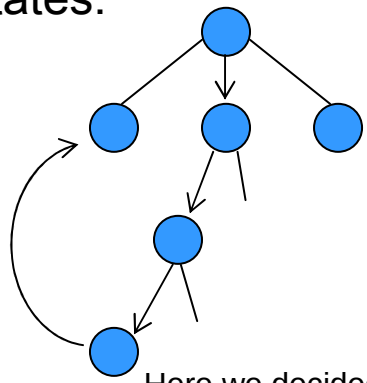
d) Indicate which actions are invoked in your search space when you move from on state to another.

on the search space

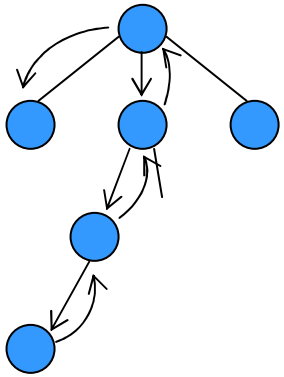
More about control strategy

When the database (DB) is relatively small, we can afford to maintain several states in memory, so we can chose from them an active one.

If we find out that we took a wrong path, we can go back to one of the previous states.



Here we decided that this path is wrong, so we go back to another state.



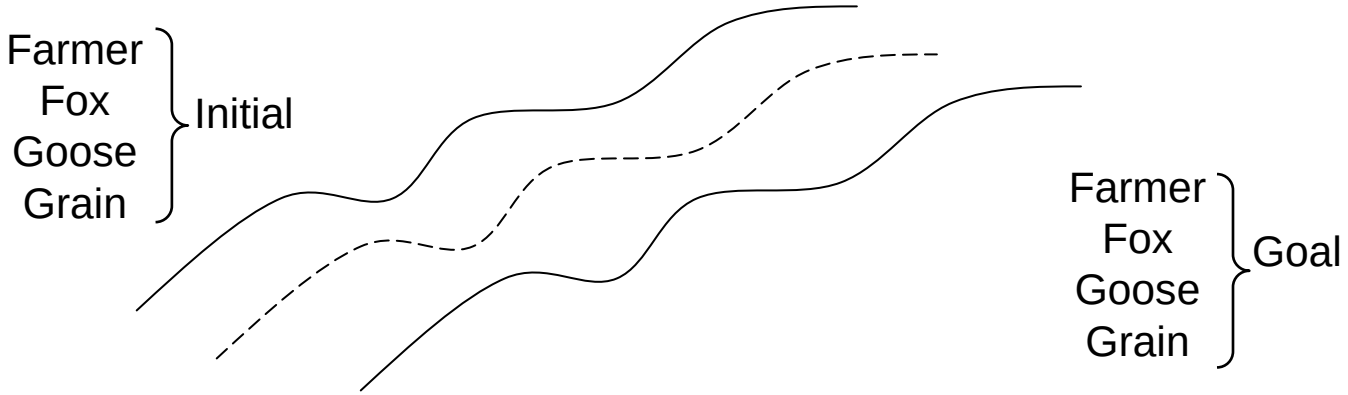
Backtracking

When the DB is large (as is usually the case) then only one state is kept. How do we correct the path?

Use backtracking—have reverse procedures that reconstitute the previous states.

Example 3 (The Farmer, Fox, Goose, Grain Puzzle)

A farmer wants to move himself, a fox, a goose and some grain across a river. His boat is tiny, he can only take one of his possessions across on one trip. An unattended fox will eat a goose, and an unattended goose will eat the grain. What should he do?



First, we decide what a state is. A difficulty is that the states are not unique; some formulations are better than others.

Here we pick a state as

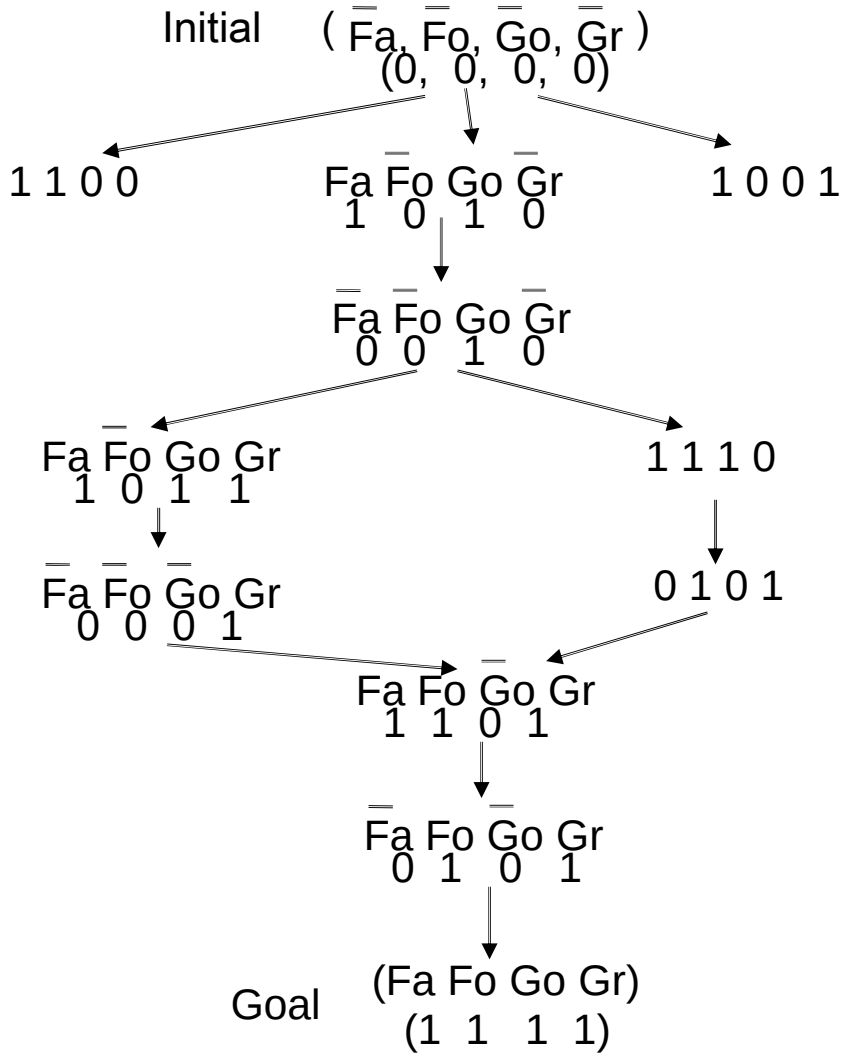
(Farmer, Fox, Goose, Grain) Or (Fa, Fo, Go, Gr)

These binary values are:

0—in one side of the river

1—in the other side of the river

State Space Search



States

Fa	Fo	Go	Gr	
0	0	0	0	✓
0	0	0	1	✓
0	0	1	0	✓
0	0	1	1	×
0	1	0	0	✓
0	1	0	1	✓
0	1	1	0	×
0	1	1	1	×
1	0	0	0	×
1	0	0	1	×
1	0	1	0	✓
1	0	1	1	✓
1	1	0	0	×
1	1	0	1	✓
1	1	1	0	✓
1	1	1	1	✓

✓ safe
× unsafe

State Space Search

Legal moves from a state to another:

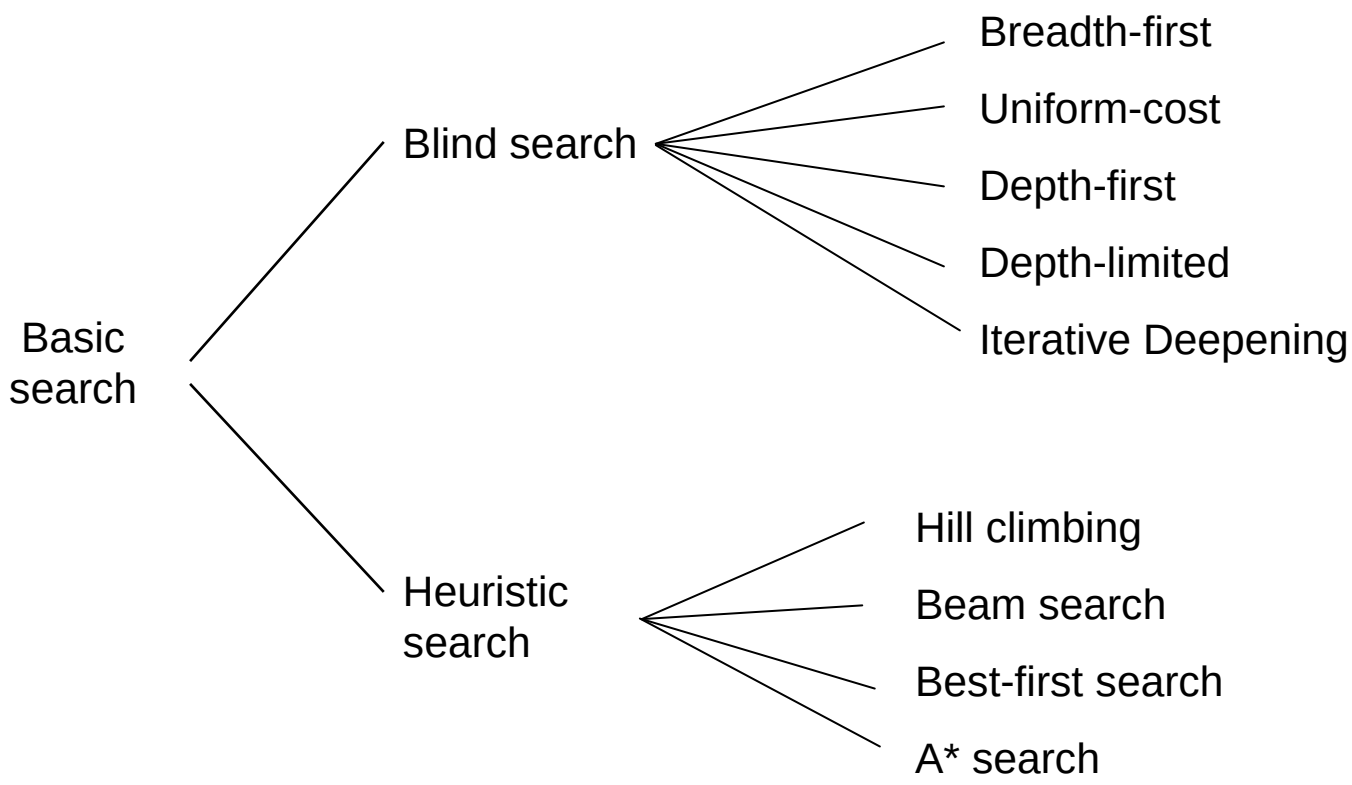
- Each move involves the farmer, &
- Farmer can carry at most one item, &
- Forbidden states

  0X11, 1X00, 011X, 100X

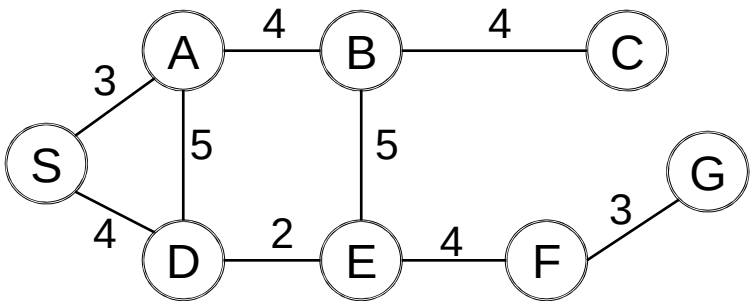
Base of this we construct the operators and the search space.

Note that the solution path is not unique. This raises the question of optimality.

Basic Search Methods



Search example



This is a highway map with cities and distances between them. The problem is to find a path (any path) from S to G (from initial/start to goal)

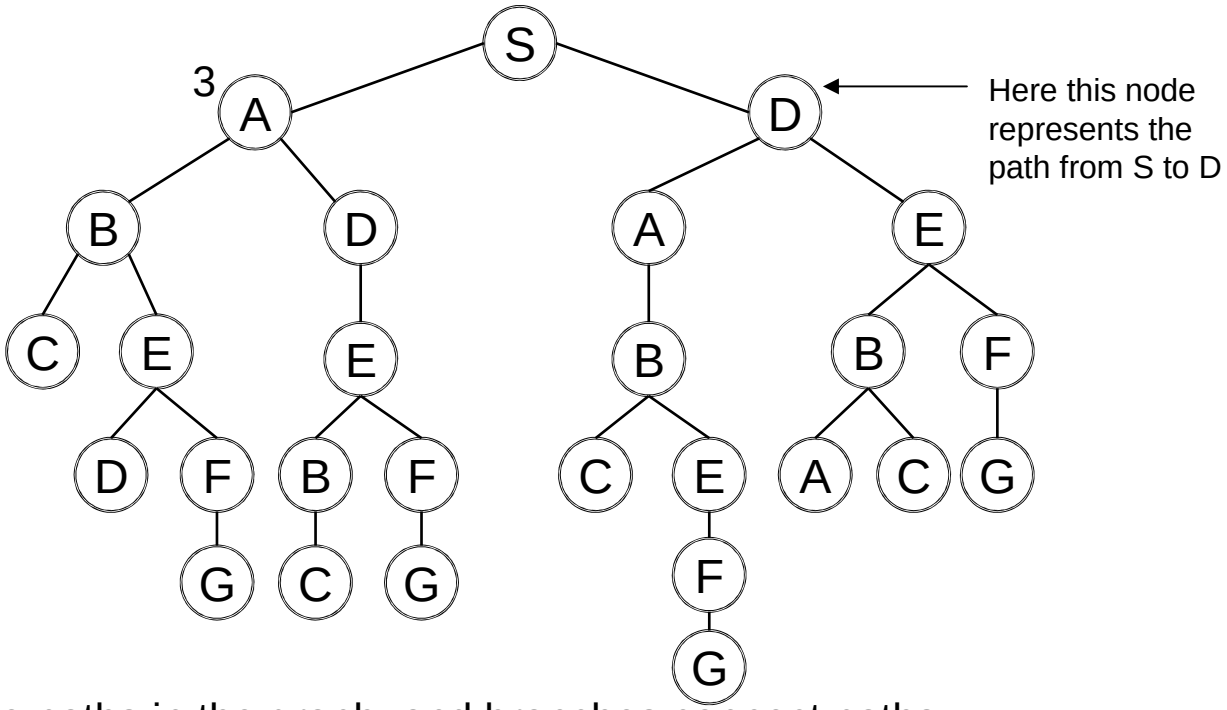
The problem of finding the shortest (optimal path) will be considered later.

Search Tree

We build a search tree that is our search space. Search trees may be very large, thus unable to completely show them, or smaller, in which case the entire tree is shown.

A search tree is not a graph.

A search tree is constructed such that paths are not redundant.



Nodes are paths in the graph, and branches connect paths.

Define: child node, parent node, root node, goal node, leaf node, ancestor node, descendent node, branching factor b.

Tree search algorithms

- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

```
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
```

```
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 to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
            only if not in the frontier or explored set
```

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

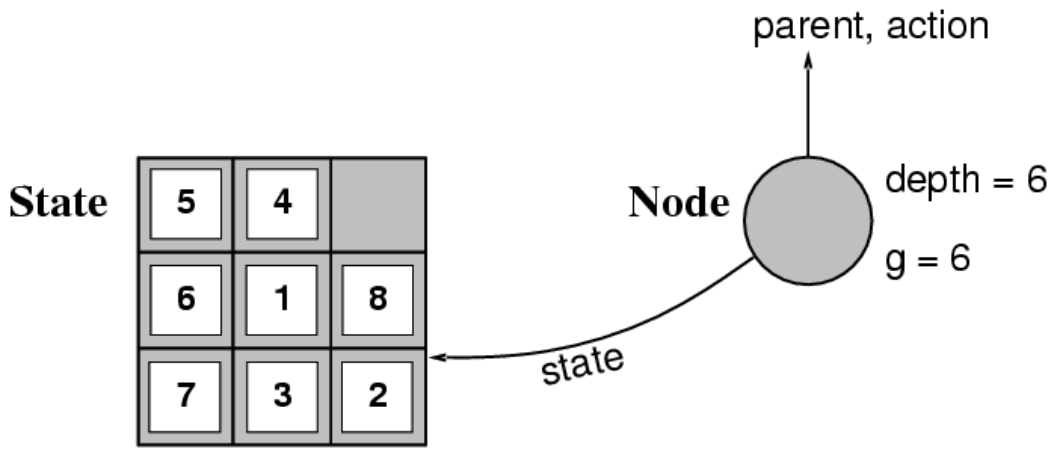
Implementation: breadth-first search

```
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)
```

Figure 3.1 Breadth-first search on a graph.

Implementation: states vs. nodes

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost $g(x)$, depth




```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
      STATE = problem.RESULT(parent.STATE, action),
      PARENT = parent, ACTION = action,
      PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

Search Methods

Depth-first search

Starting from root node pick one child at every node visited and move forward.

The search may be long, but it finds a solution

Breadth-first search

Searches all nodes at a level n before moving to level n+1.

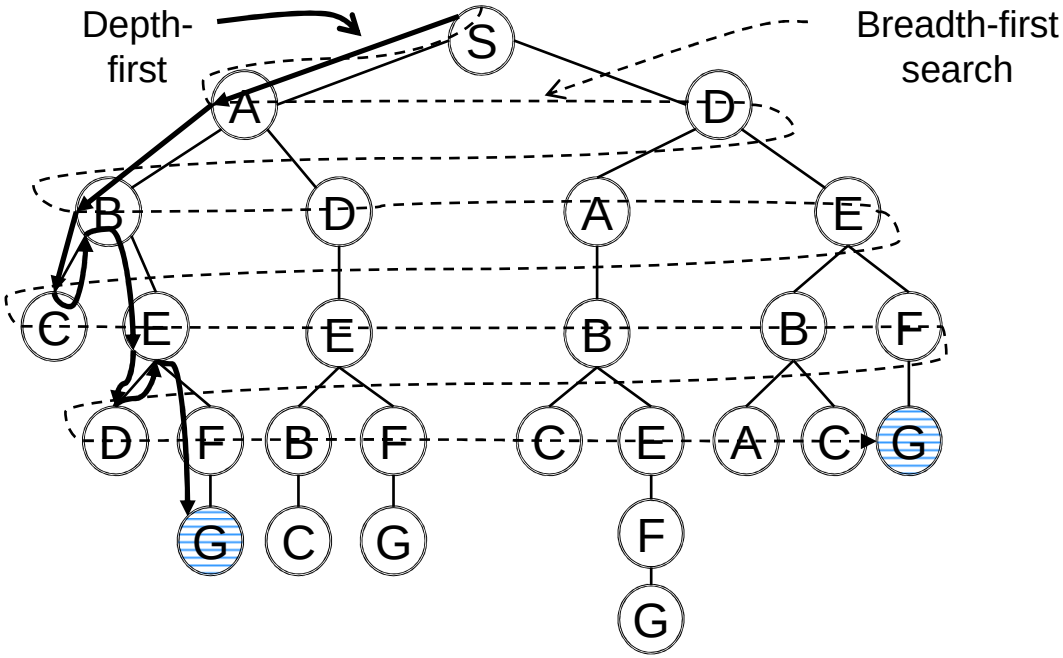
The search is usually very long but it finds a solution.

Nondeterministic search

Moves randomly into the search tree.

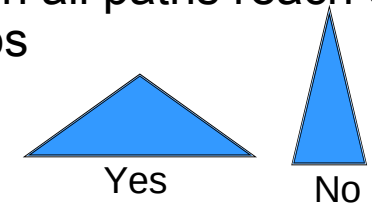
Note: These search methods do not use information about the problem. Thus they are called uninformed or blind search methods.

Search Tree



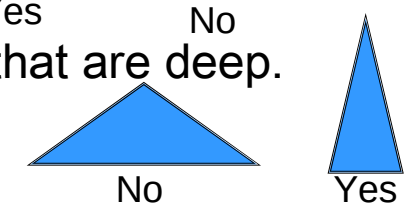
Depth-first is recommended when all paths reach dead ends, or reach the goal in reasonable number of steps

d—depth of tree is small



Breadth first search is better for trees that are deep.

Not good for large b.



Search strategies

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)

Properties of breadth-first search

- Complete? Yes (if b is finite)
- Time? $b + b^2 + b^3 + \ldots + b^d = O(b^d)$
- Space? $O(b^d)$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)

- Space is the bigger problem (more than time)

Uniform-cost search

- Expand least-cost unexpanded node
- Implementation:
  - *frontier* = priority queue ordered by path cost *g(n)*
- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost ≥ ε
- Time? # of nodes with $g \leq$ cost of optimal solution, $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$ where C^* is the cost of the optimal solution
- Space? # of nodes with $g \leq$ cost of optimal solution, $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$
- Optimal? Yes – nodes expanded in increasing order of *g(n)*

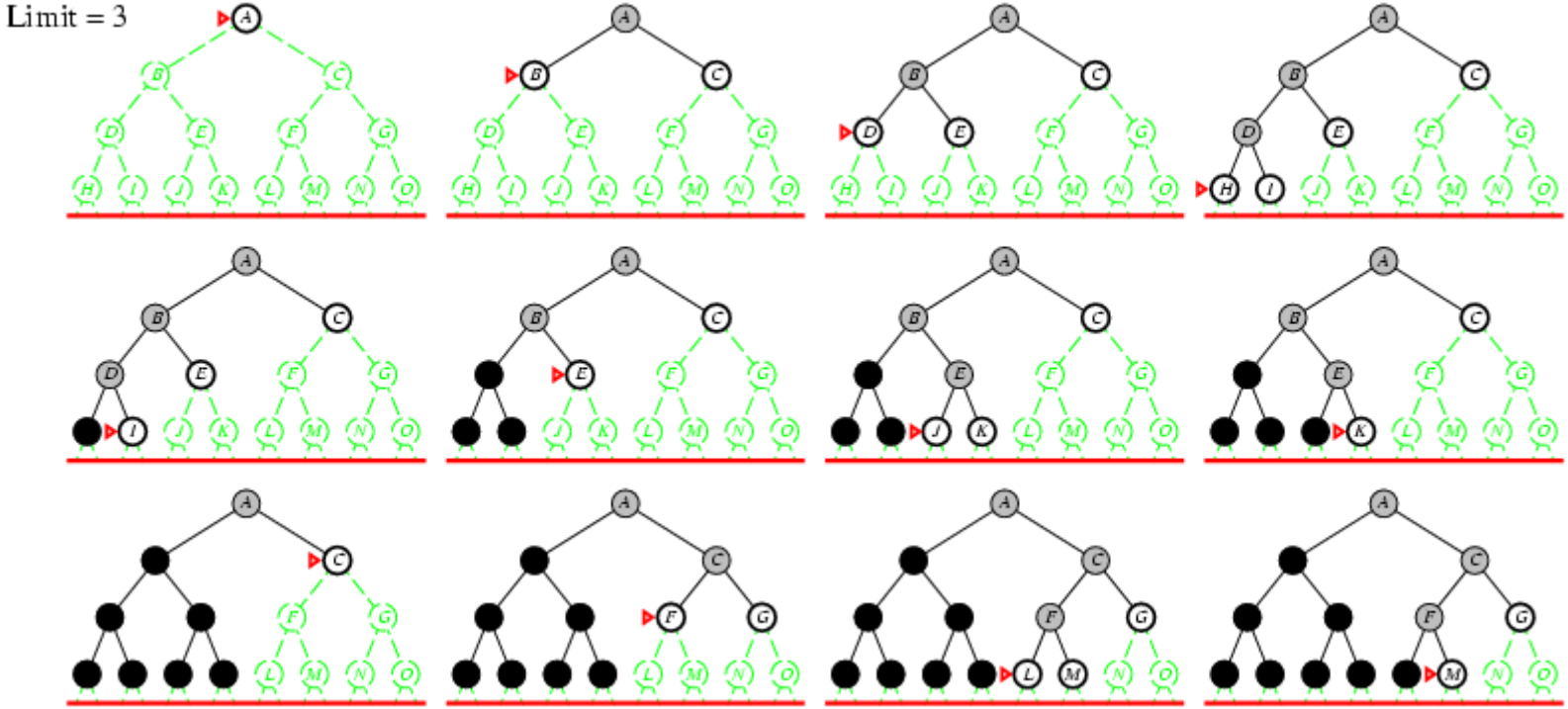
Note:

- Complexities are not computed as function of b and d, instead of optimal cost C*
- Uniform-cost search examines all the nodes at the goal's depth to see if one has lower cost.

Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
    → complete in finite spaces
- Time? $O(b^m)$: terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, i.e., linear space!
- Optimal? No

Iterative deepening search l =3



Iterative deepening search

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

$$N_{DLS} = b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

$$N_{IDS} = d\ b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d = O(b^d)$$

- For b = 10, d = 5,

  N_{DLS} = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110

  N_{IDS} = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450

- Overhead = (123,450 - 111,110)/111,110 = 11%

IDS is iterative-deepening search
DLS is depth-limited search

Properties of iterative deepening search

- Complete? Yes
- Time? $d\ b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? O(bd)
- Optimal? Yes, if step cost = 1

Summary of algorithms

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

Heuristic Search

Is used to curb the combinational explosion. For a tree with depth d and branching factor b the total number of nodes is b^d (exponential with depth d).

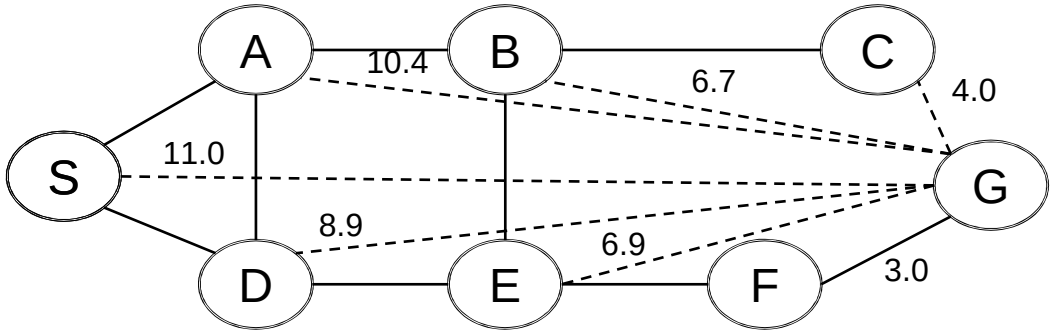
Heuristic means to discover; in AI it means improving problem-solving performance. It does this by limiting the search for solutions in large problems.

Heuristics do not guarantee optimal solutions, in fact they do not guarantee any solutions at all.

So the problem is to identify additional information about the problem—called heuristic function—that improves search efficiency.

With this additional information, heuristic search methods are variations of blind methods.

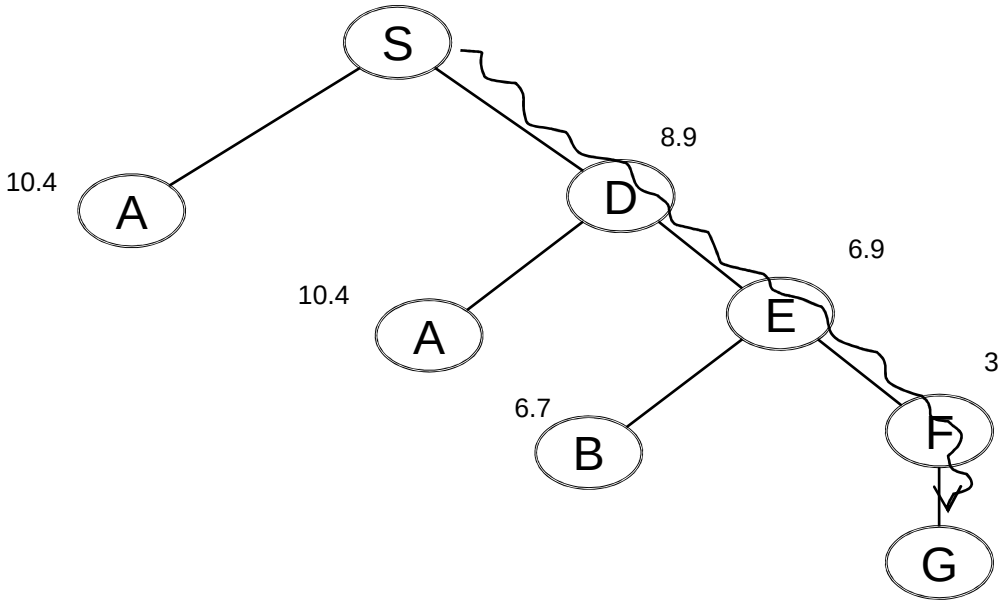
Heuristic Search



Dotted lines indicate direct distances between cities and goal G. This is a heuristic function which is used as an evaluation function.

Hill Climbing

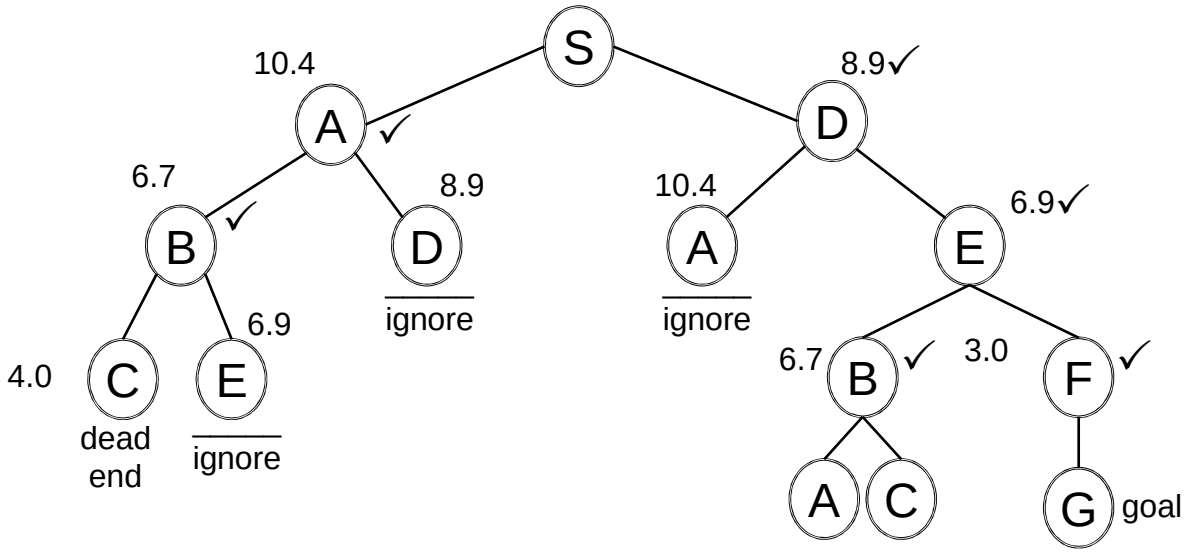
Hill climbing is a depth-first search methods that selects the child node according with a heuristic measurement; in this case we pick the node which has the smallest distance to the goal.



In this case, by coincidence it happened to be the optimum path.

Beam Search

Beam search is like breadth-search, but it moves forward only to a limited number of nodes (w). The other nodes are ignored.



Note that only the best w = 2 nodes are expanded from an entire level n.

Best-First search – expands the best partial path.

The figure coincides with hill-climbing (It is also called best promising).

Optimal Search

Branch and bound

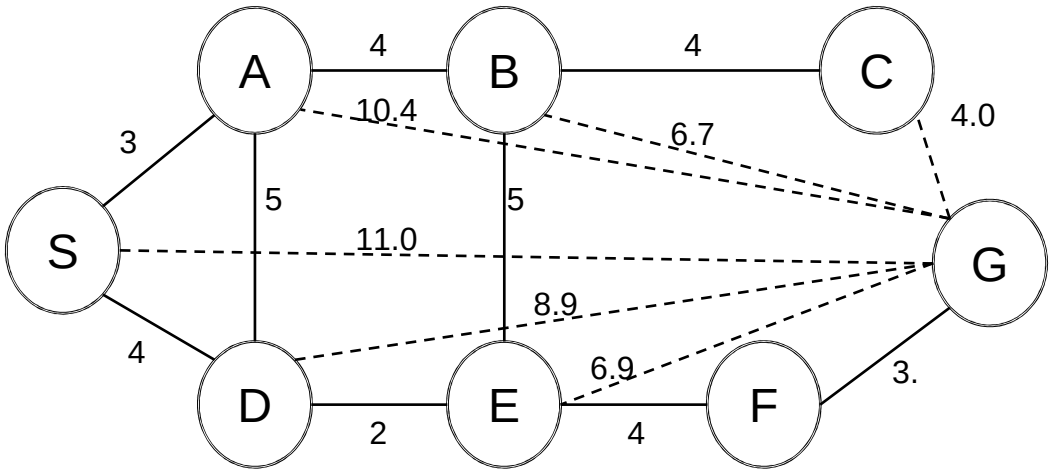
Dynamic programming

A*

Optimal search finds the best path (best in the sense of shortest path).

Note that optimality may be interpreted in other ways—for example to minimize search effort—or computation cost.

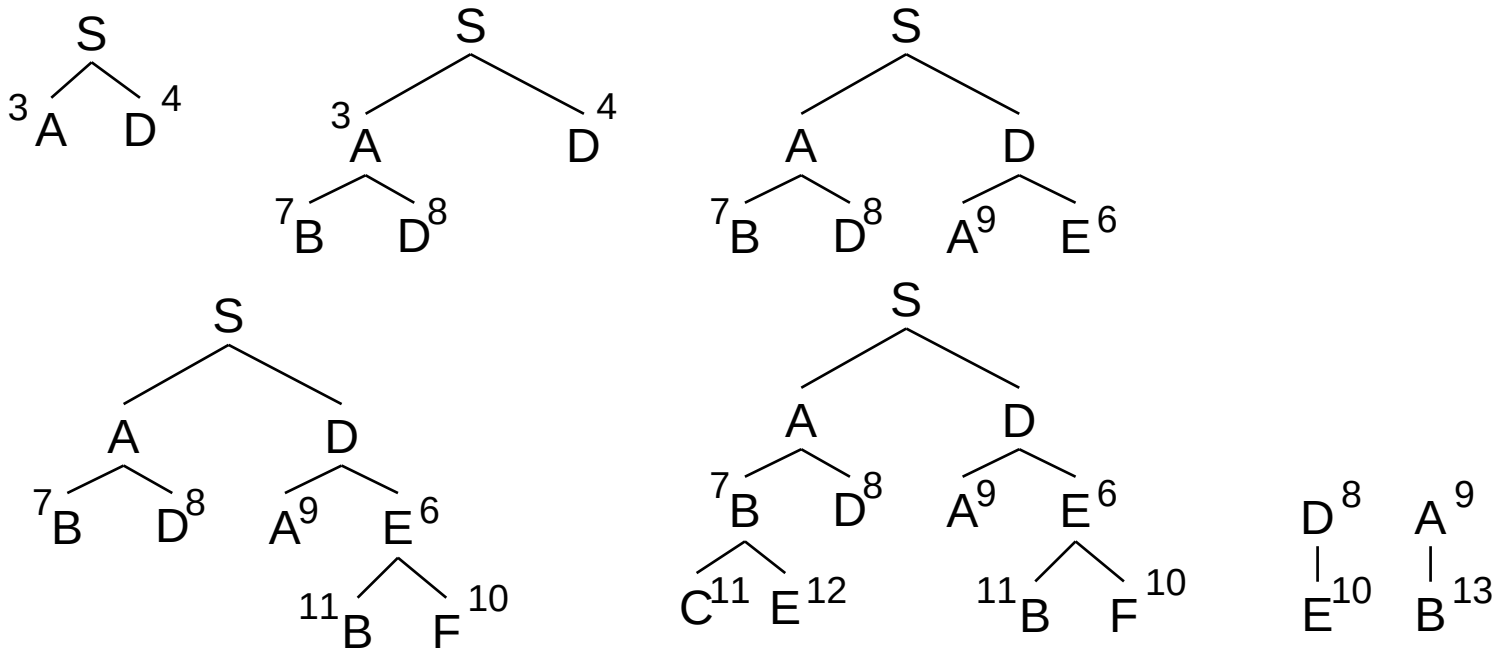
A trivial method is to find all solutions, and then to select the best. Obviously, this is too expansive for actual trees.



Branch-and-Bound

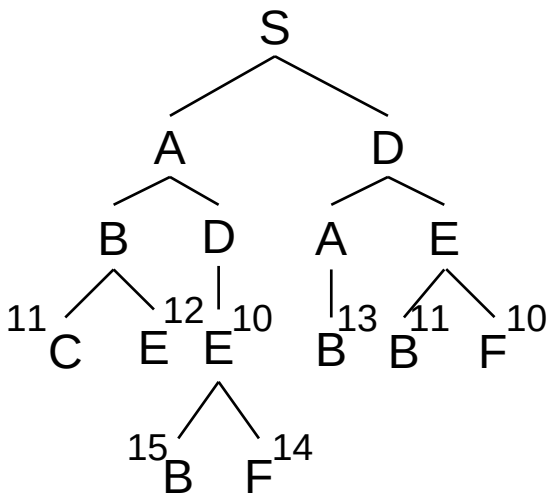
Idea: Expand the shortest path first one level creating all possible branches. Then, expand the remaining paths and determine which is the new path to be expanded. Continue until the goal is reached.

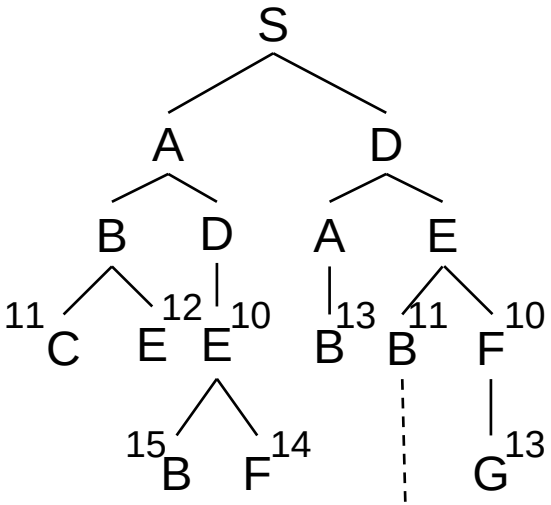
Since the shortest path is extended first, it is likely that the first path reaching the goal is the optimal path.



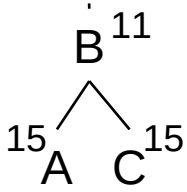
Note that all likely paths need to be expanded till cost is higher than complete path (solution).

Branch-and-Bound





Although we reached the goal, we have to see if another path is not shorter.



Since these are longer than the goal the search stops here.

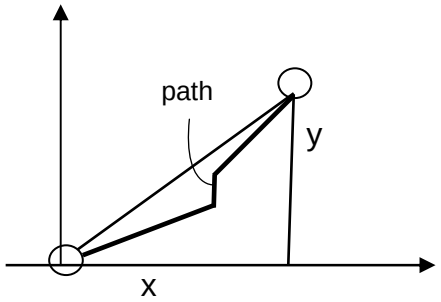
Use Underestimates to Improve Efficiency

We guess the remaining distance.

e(total page) = d(already traveled) + e(remaining distance)

where: e —estimated distance
d —known distance

The estimate has to be an underestimation (like straight line), because if it is an overestimation along the optimal path, sometimes it may cause the algorithm to move away from the optimal path. In other words, the optimality is not guaranteed.

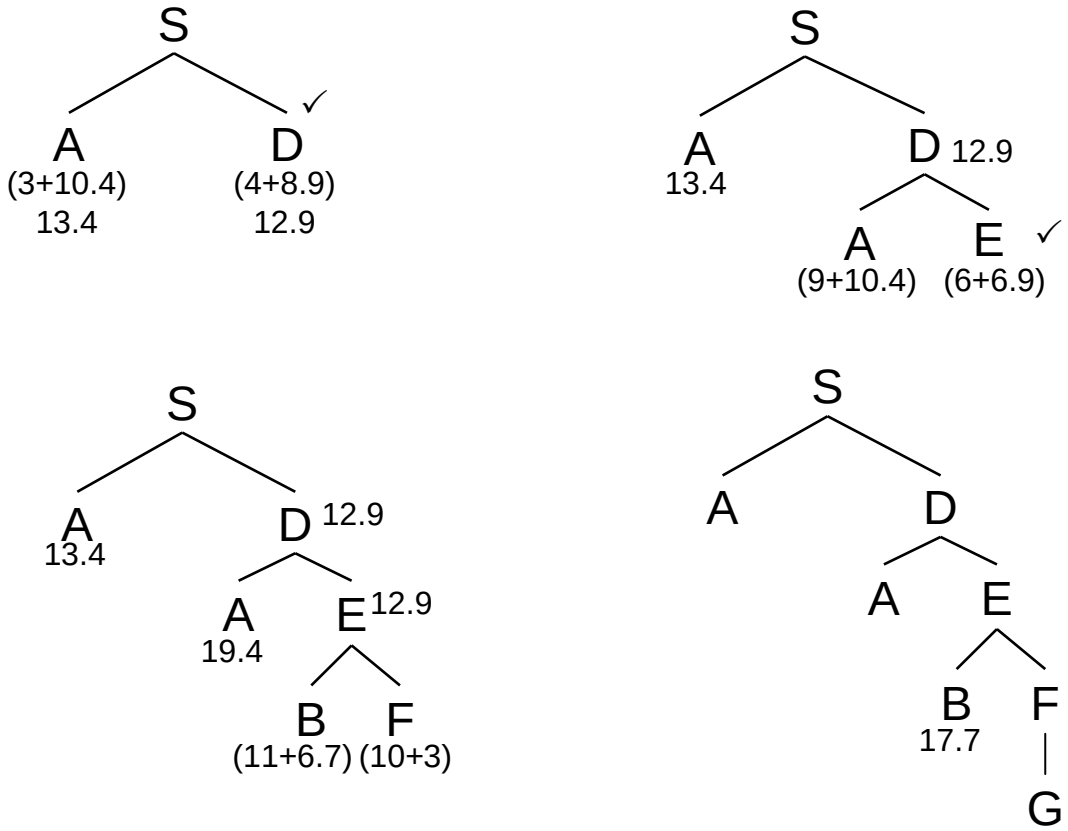


x + y – is overestimate

straight line is underestimate

u(total path) = d(already traveled) + u(remaining distance)

Use Underestimates to Improve Efficiency

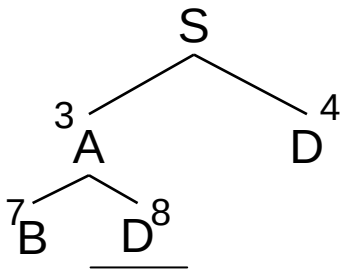


The closer the underestimate is to the actual distance the better (fewer nodes to expand).

If underestimate is 0 then is like branch-and-bound.

Redundant Paths

Expand as in branch-and-bound, but stop expanding nodes that have already been found and have lower cost.



Stop since there is another D with smaller value