G52APT
AI Programming Techniques

Lecture 10: Breadth first search in Prolog

Brian Logan
School of Computer Science
bsl@cs.nott.ac.uk

Outline of this lecture

- recap: breadth first search
- breadth first search in Prolog
- agenda based search
- preferred solutions and path costs
- uniform-cost search

Recap: definition of a search problem

- a *search problem* is defined by:
  - a *state space* (i.e., an initial state or set of initial states and a set of operators)
  - a *set of goal states* (listed explicitly or given implicitly by means of a property that can be applied to a state to determine if it is a goal state)
- a *solution* is a path in the state space from an initial state to a goal state

Recap: exploring the state space

- *search* is the process of exploring the state space to find a solution
- exploration starts from the initial state
- the search procedure applies operators to the initial state to generate one or more new states which are hopefully nearer to a solution
- the search procedure is then applied recursively to the newly generated states
- the procedure terminates when either a solution is found, or no operators can be applied to any of the current states

Recap: search trees

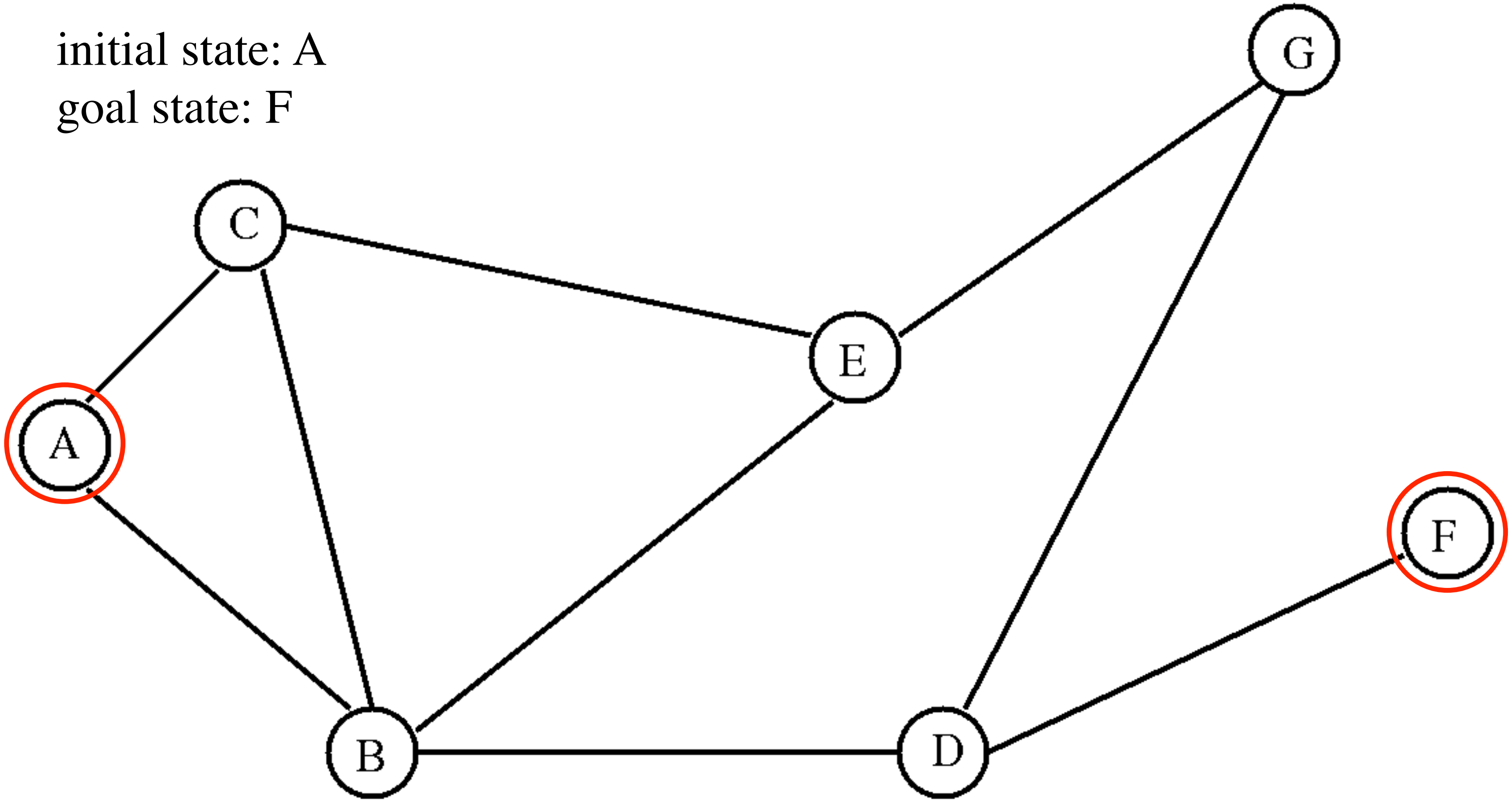
- the part of the state space that has been explored by a search procedure can be represented as a *search tree*
- nodes in the search tree represent *paths* from the initial state (i.e., partial solutions) and edges represent operator applications
- the process of generating the children of a node by applying operators is called *expanding* the node
- the branching factor of a search tree is the average number of children of each non-leaf node
- if the branching factor is b, the number of nodes at depth d is b^d

Breadth-first search

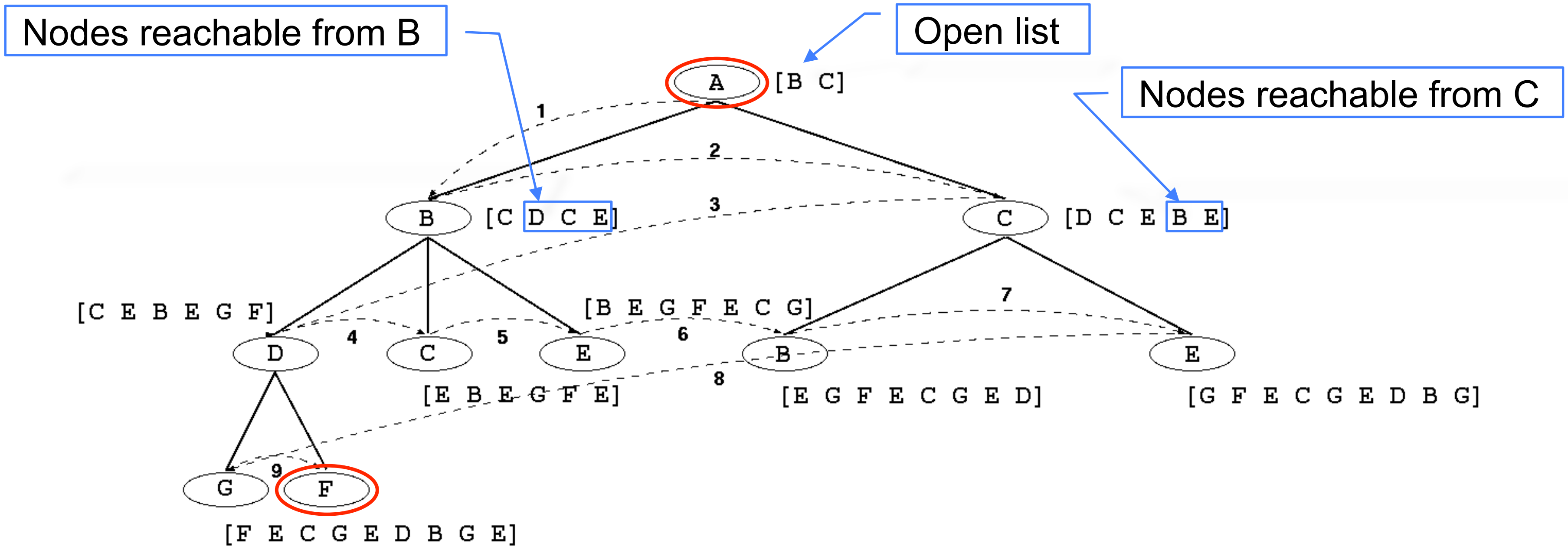
- proceeds level by level down the search tree
- first explores all paths of length 1 from the root node, then all paths of length 2, length 3 etc.
- starting from the root node (initial state) explores all children of the root node, left to right
- if no solution is found, expands the first (leftmost) child of the root node, then expands the second node at depth 1 and so on …

Example: simple route planning

initial state: A
goal state: F



Example: breadth-first search



Properties of breadth-first search

- breadth-first search is *complete* (even if the state space is infinite or contains loops)
- it is guaranteed to find the solution requiring the smallest number of operator applications (an *optimal solution* if cost is a non-decreasing function of the depth of a node)
- time and space complexity is $O(b^d)$ where d is the depth of the shallowest solution
- severely space bound in practice, and often runs out of memory very quickly

Comparison with depth-first (tree) search

- space complexity is O(bm) where b is the branching factor and m is the maximum depth of the tree
- time complexity is O(b^m)
- not complete (unless the state space is finite and contains no loops)—we may get stuck going down an infinite branch that doesn't lead to a solution
- even if the state space is finite and contains no loops, the first solution found by depth-first search may not be the shortest

Search in Prolog

- if there is a solution, a search procedure should return it (completeness)
- if the state space is finite, a search procedure should return all solutions on backtracking
- if the state space is finite and there are no solutions, a search procedure should return 'no'
- if the state space is infinite, a search procedure may not terminate unless there is a depth bound – we may wish to signal this type of failure differently

Optimal search in Prolog

- what should these criteria mean for breadth-first search (and more generally for search algorithms which are optimal)?

- assuming the state space (and branching factor) is finite, we could return:

  a) an optimal solution, and no additional solutions on backtracking (on the grounds that an optimal search algorithm should return only optimal solutions, and any optimal solution is as good as any other)

  b) an optimal solution, and all other optimal solutions on backtracking

  c) an optimal solution, and all other solutions on backtracking (e.g., in order of increasing depth/cost) – should an optimal algorithm return non-optimal solutions?

Criteria for breadth-first search in Prolog

- if path cost is an increasing function of depth (i.e., shortest solutions are 'optimal'):
  - option (a) seems the most reasonable choice if 'optimal' means *optimal* – i.e., it's impossible to do better
  - option (b) is reasonable if 'optimal' means *good*, e.g., a journey planner may wish to offer a user all alternative journeys which involve the minimum number of changes, and leave it to the user to choose which one they prefer
- if path cost is a general function, option (c) seems the most reasonable
- when implementing a *general* breadth-first search algorithm intended for use on a range of problems, all we can do is choose one of these options and document what the algorithm does

Implementing search in Prolog

- to implement search in Prolog we need to decide ...
- how to represent the search problem:
  - states – what properties of states are relevant
  - operators – including the applicability of operators
  - goals – should these be represented as a set of states or a test
- how to represent the search tree and the state of the search:
  - paths – what information about a path is relevant (cost(s) etc)
  - nodes – parent, children, depth in the tree etc.
  - open/closed lists

Breadth first search

```
:- use_module(library(lists)).
% bfs(?initial_state, ?goal_state, ?solution)
% does not include cycle detection
bfs(X,Y,P) :-
    bfs_a(Y,[n(X,[])],R),
    reverse(R,P).

bfs_a(Y,[n(Y,P)|_],P).
bfs_a(Y,[n(S,P1)|Ns],P) :-
    findall(n(S1,[A|P1]),s(S,S1,A),Es),
    append(Ns,Es,O),
    bfs_a(Y,O,P).

% s(?state, ?next_state, ?operator).
s(a,b,go(a,b)).
s(a,c,go(a,c)).
etc...
```

Representation of states & operators

- **states** are represented by atoms in arguments to the *s*/3 successor relation and the *n*/2 terms

- **operators** are represented by complex terms (*go*/2 terms) that represent an action that transforms one state into another, e.g., `go (a,b)`

- **applicability of operators** is determined by the explicit current state and the existence of an appropriate successor of the current state

- **goal** is explicit and is an argument to *bfs*/3

Representation of nodes & paths

- **path**s (and solutions) are represented by lists of operator terms, e.g., `[go(b,d),go(a,b)]` (in reverse order)

- **nodes** are explicit and represented by *n*/2 terms, where the first argument is a state and the second argument is a path to that state

- the **open list** is explicitly represented by a list of *n*/2 terms

- **closed list** is not used, but can be explicitly represented by a list of states (see later)

Properties of *bfs_a*/3

- *bfs_a*/3 is *complete* even if the state space is infinite or contains loops (assuming a finite branching factor)
- guaranteed to find the solution requiring the smallest number of operator applications
- time and space complexity is $O(b^d)$ where d is the depth of the shallowest solution
- on backtracking
  - may return non-optimal solutions
  - doesn't terminate if the state space contains loops

Breadth first search with closed list

```
% bfs(?initial_state, ?goal_state, ?solution)
bfs(X,Y,P) :-
    bfs_b(Y,[n(X,[])],[],R),
    reverse(R,P).

bfs_b(Y,[n(Y,P)|_],_,P).
bfs_b(Y,[n(S,P1)|Ns],C,P) :-
    findall(n(S1,[A|P1]),
            (s(S,S1,A), \+ member(S1,C)),
           Es),
    append(Ns,Es,O),
    bfs_b(Y,O,[S|C],P).
```

Properties of *bfs_b*/4

- completeness, optimality and complexity as for *bfs_a*/3
- on backtracking
  - may return non-optimal solutions
  - terminates if the state space is finite (even if it contains loops)

Breadth first search with closed list 2

```
% bfs(?initial_state, ?goal_state, ?solution)
bfs(X,Y,P) :-
    bfs_c(Y,[n(X,[])],[n(X,[])],R),
    reverse(R,P).

bfs_c(Y,[n(Y,P)|_],_,P).
bfs_c(Y,[n(S,P1)|Ns],C,P) :-
    findall(n(S1,[A|P1]),
            (s(S,S1,A), \+ member(n(S1,_),C)),
            Es),
    append(Ns,Es,O),
    append(C,Es,C1),
    bfs_c(Y,O,C1,P).
```

Properties of *bfs_c*/4

- completeness, optimality and complexity as for *bfs_a*/3
- on backtracking
  - does not return alternative optimal (shortest) solutions (i.e., returns only the first optimal solution found)
  - terminates if the state space is finite (even if it contains loops)

Breadth first search with closed list 3

```
% bfs(?initial_state, ?goal_state, ?solution)
bfs(X,Y,P) :-
    bfs_d(Y,[n(X,[])],[],R),
    reverse(R,P).

bfs_d(Y,[n(Y,P)|_],_,P).
bfs_d(Y,[n(S,P1)|Ns],C,P) :-
    length(P1,L),
    findall(n(S1,[A|P1]),
            (s(S,S1,A),
             \+ (member(n(S1,P2),Ns), length(P2,L)),
             \+ member(S1,C)),
            Es),
    append(Ns,Es,O),
    bfs_d(Y,O,[S|C],P).
```

Properties of *bfs_d*/4

- completeness, optimality and complexity as for *bfs_a*/3
- on backtracking
  – returns all optimal (shortest) solutions
  – terminates if the state space is finite (even if it contains loops)

Open list

- the addition of an open list (or queue, or agenda) is the basis for many other forms of search

- different search algorithms are defined by the order in which nodes are inserted in the open list

  – depth first search

  – uniform cost search

  – greedy search

  – A* search

  – etc.

Aside: depth first search with an open list

```
:- use_module(library(lists)).
% dfs(?initial_state, ?goal_state, ?solution)
% does not include cycle detection
dfs(X,Y,P) :-
    dfs_a(Y,[n(X,[])],R),
    reverse(R,P).

dfs_a(Y,[n(Y,P)|_],P).
dfs_a(Y,[n(S,P1)|Ns],P) :-
    findall(n(S1,[A|P1]),s(S,S1,A),Es),
    append(Es,Ns,O),
    dfs_a(Y,O,P).

% s(?state, ?next_state, ?operator).
s(a,b,go(a,b)).
s(a,c,go(a,c)).
etc...
```

Extending the problem definition

- a *search problem* is defined by:
  - a *state space* (i.e., an initial state or set of initial states and a set of operators)
  - a *set of goal states* (listed explicitly or given implicitly by means of a property that can be applied to a state to determine if it is a goal state)
- a *solution* is *any* path in the state space from an initial state to a goal state

Preferred solutions

- one solution is often *preferable* to another—e.g., we may prefer paths with fewer or less costly actions
- in the route planning problem, we might prefer solutions which
  - minimise the distance travelled
  - the time taken to reach the goal
  - the number of cities (changes) if we are travelling by train
  - the monetary cost (of fuel or train tickets etc)
  - or some *combination* of these and other factors …

Path cost

- a path cost function, $g(n)$, assigns a *cost* to a path n and can be used to rank alternative solutions

- if all operators have the *same cost* (e.g, moves in chess) the cost is simply the number of operator applications

- if different operators have *different costs* (e.g, money, time etc) the path cost is sum of the costs of all the operator applications in the path

- a search procedure which is guaranteed to find a *least cost solution* (if a solution exists) is said to be *optimal*

Comparison

- **breadth-first search** is complete and guaranteed to find an *optimal solution* if cost is a non-decreasing function of the depth of a node— e.g., if all operators have the same cost

- **iterative deepening search** is complete and there is a solution at some *finite* depth; guaranteed to find an optimal solution if cost is a non-decreasing function of the depth of a node

- **depth limited search** is complete if complete if there is a solution within the depth bound $d \leq l$, but is not guaranteed to find an optimal solution

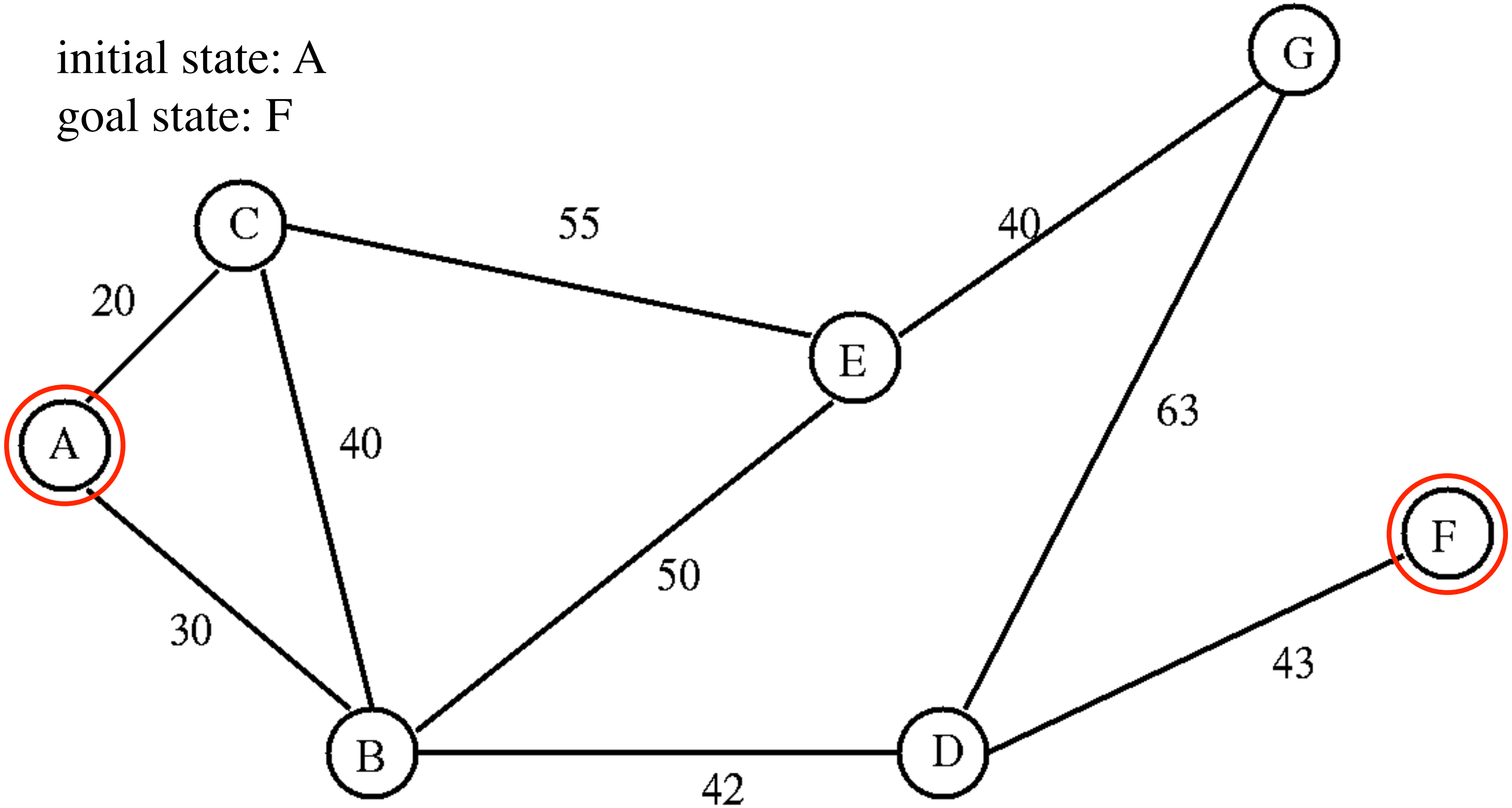
- **depth-first search** is complete if the state space is finite an contains no loops, but is not guaranteed to find a optimal solution

Uniform-cost search

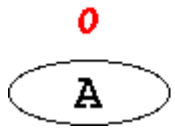
- breadth-first search finds the *shallowest* goal state
- for a *general* path cost function this may not always be the least cost solution
- *uniform-cost search* expands leaf nodes in order of cost (as measured by the path cost $g(n)$)
- expands the root node, then the lowest cost child of the root node, then the lowest cost unexpanded node etc.

Example: simple route planning

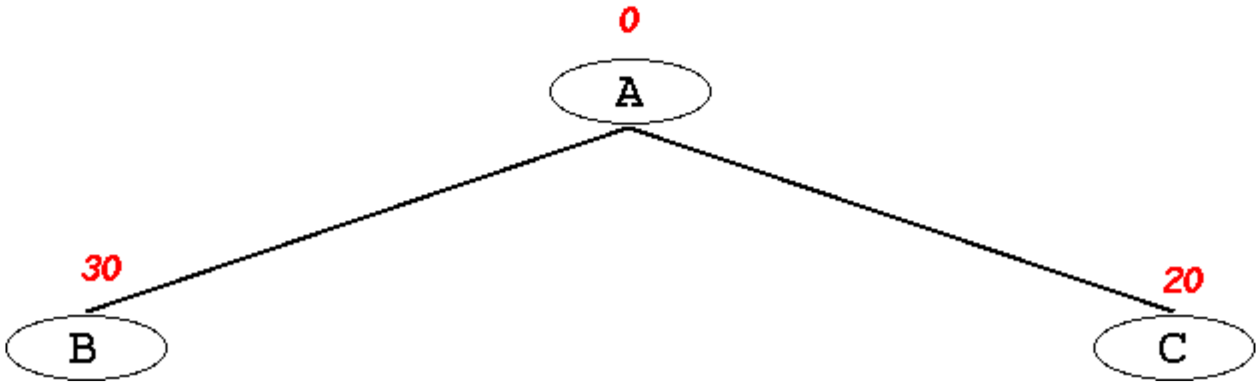
initial state: A
goal state: F



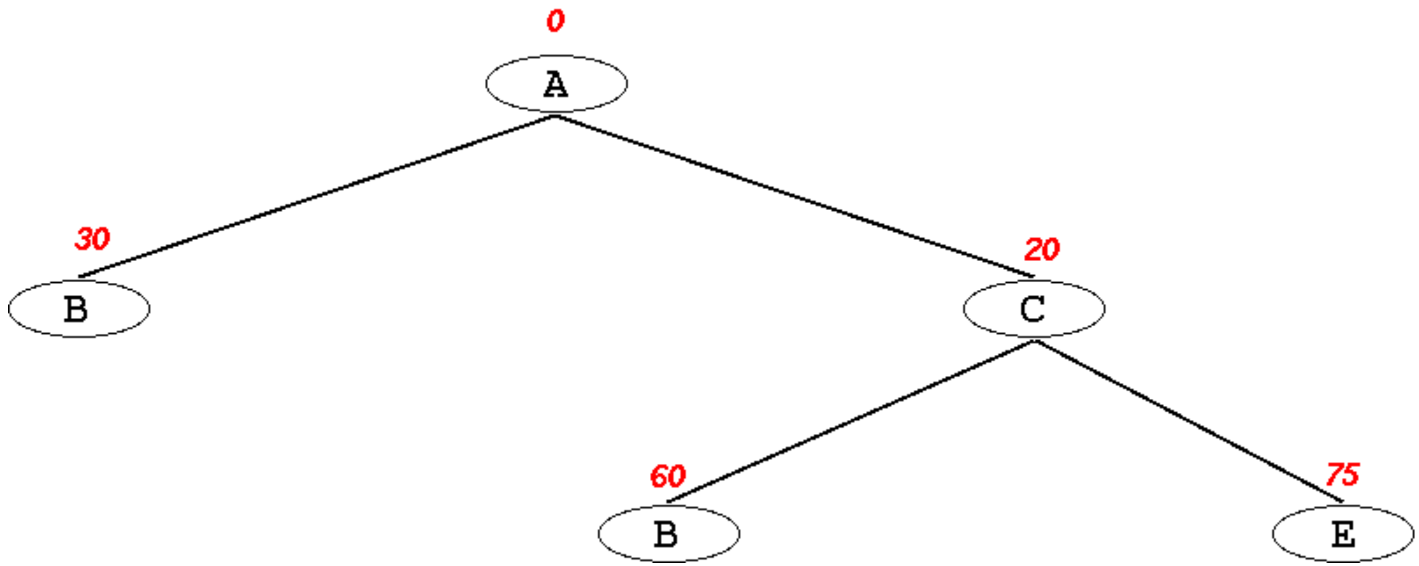
Example: uniform-cost search



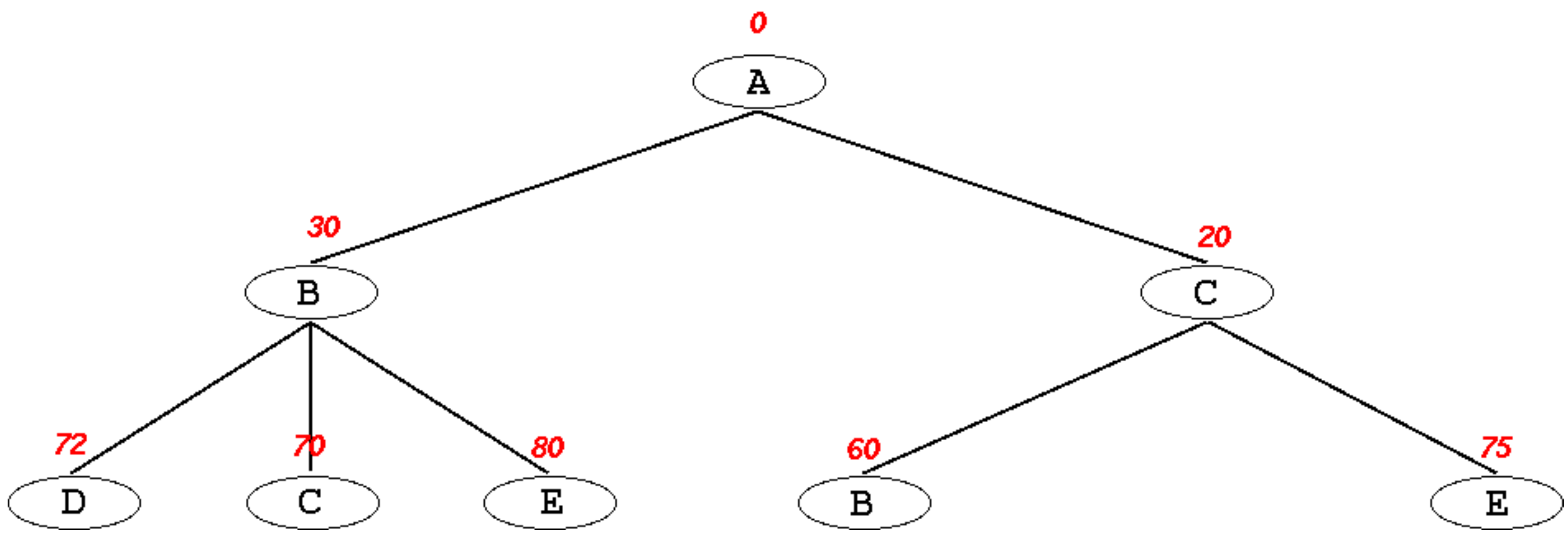
Example: uniform-cost search



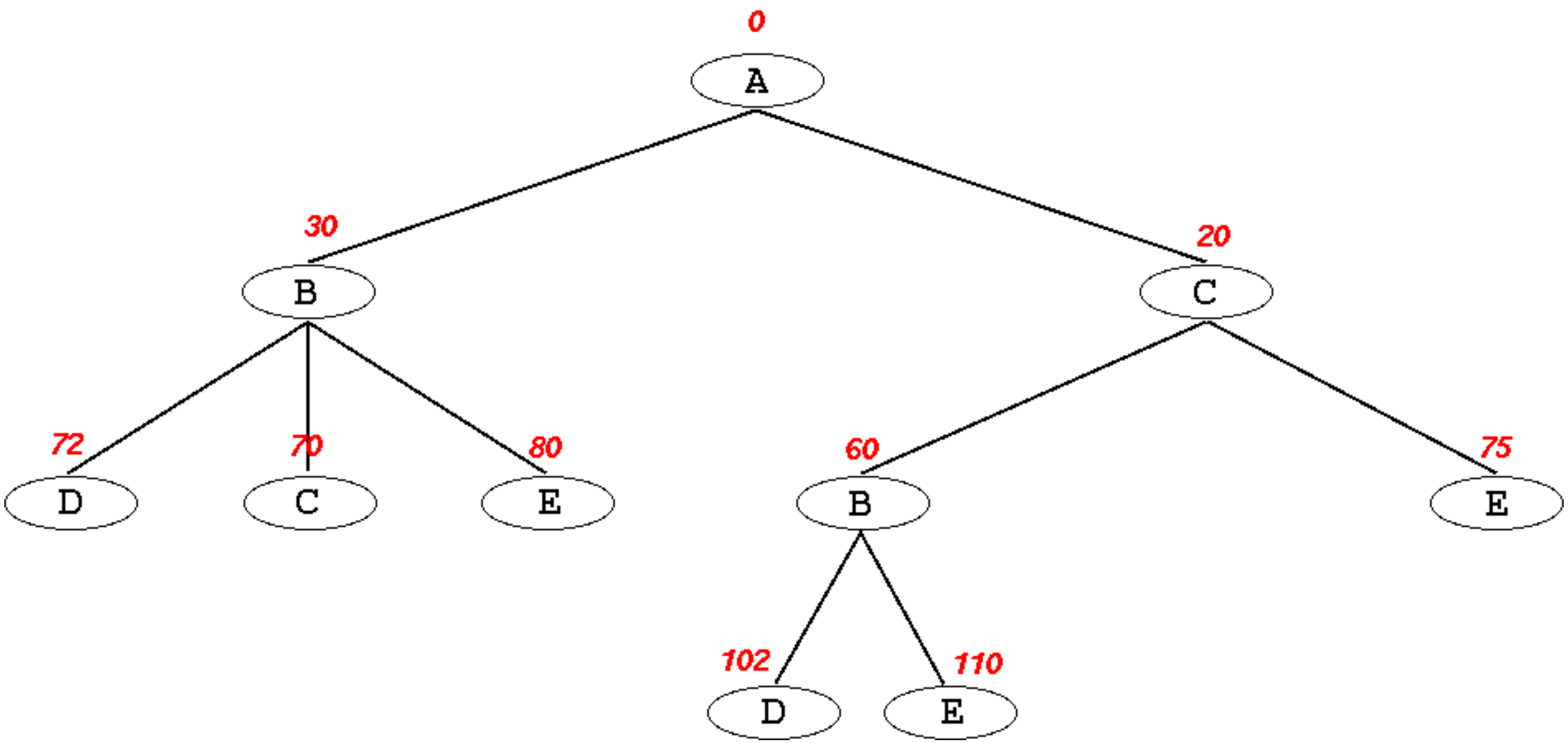
Example: uniform-cost search



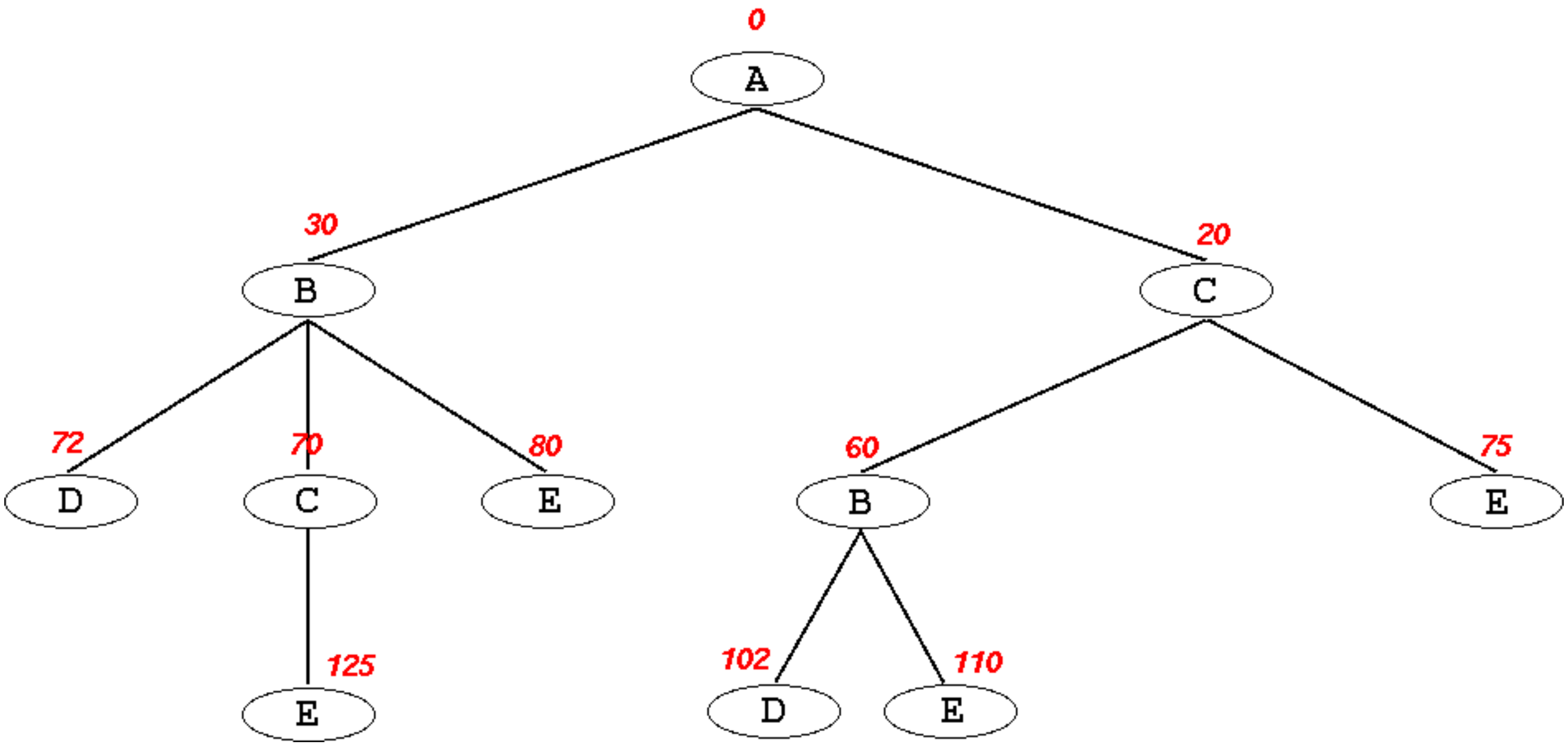
Example: uniform-cost search



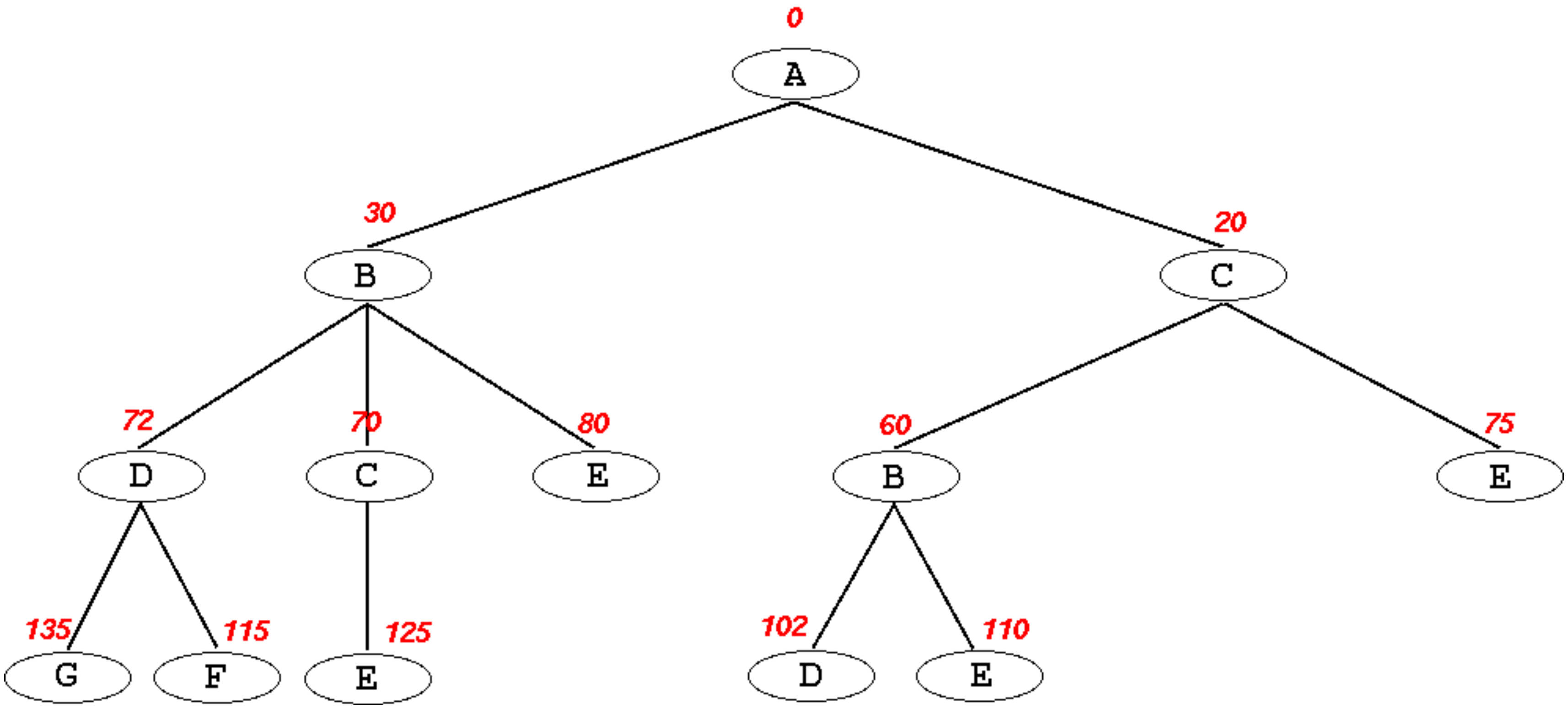
Example: uniform-cost search



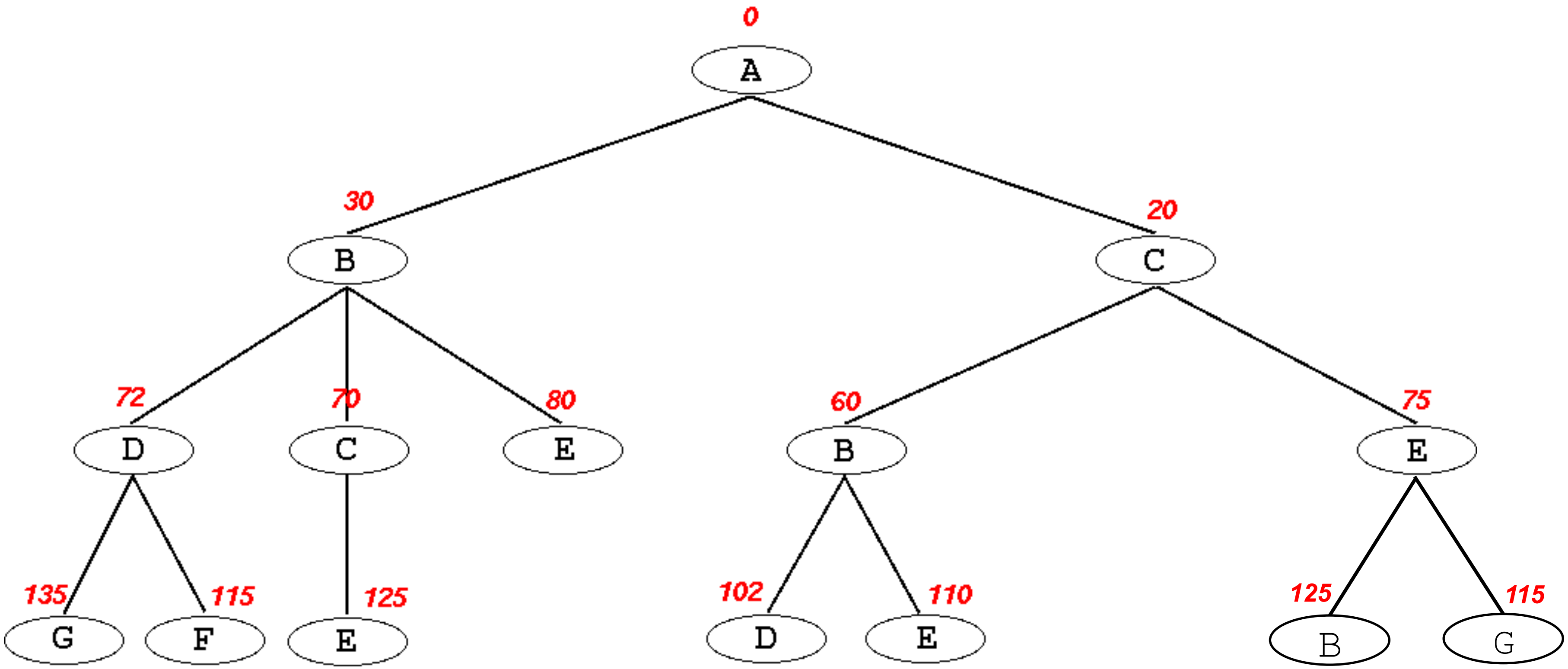
Example: uniform-cost search



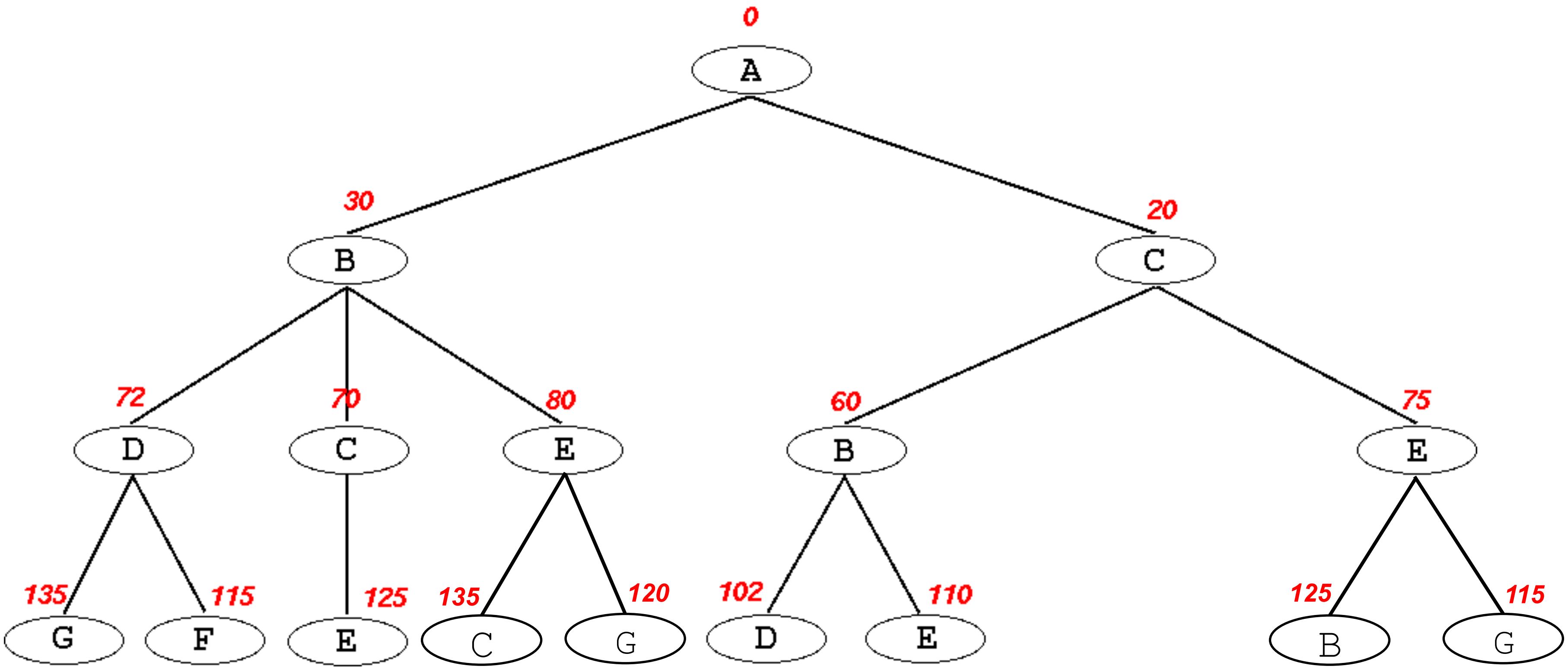
Example: uniform-cost search



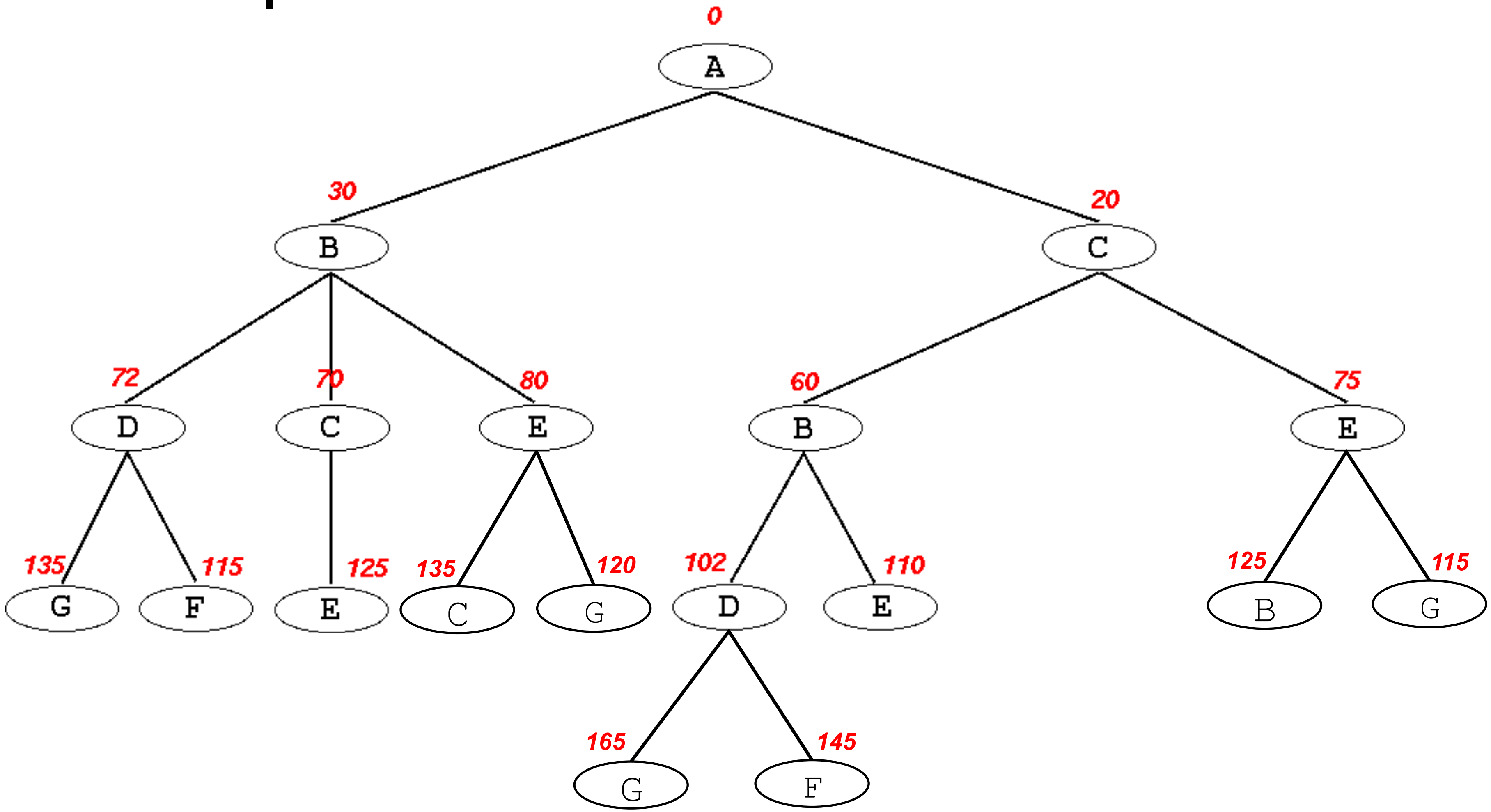
Example: uniform-cost search



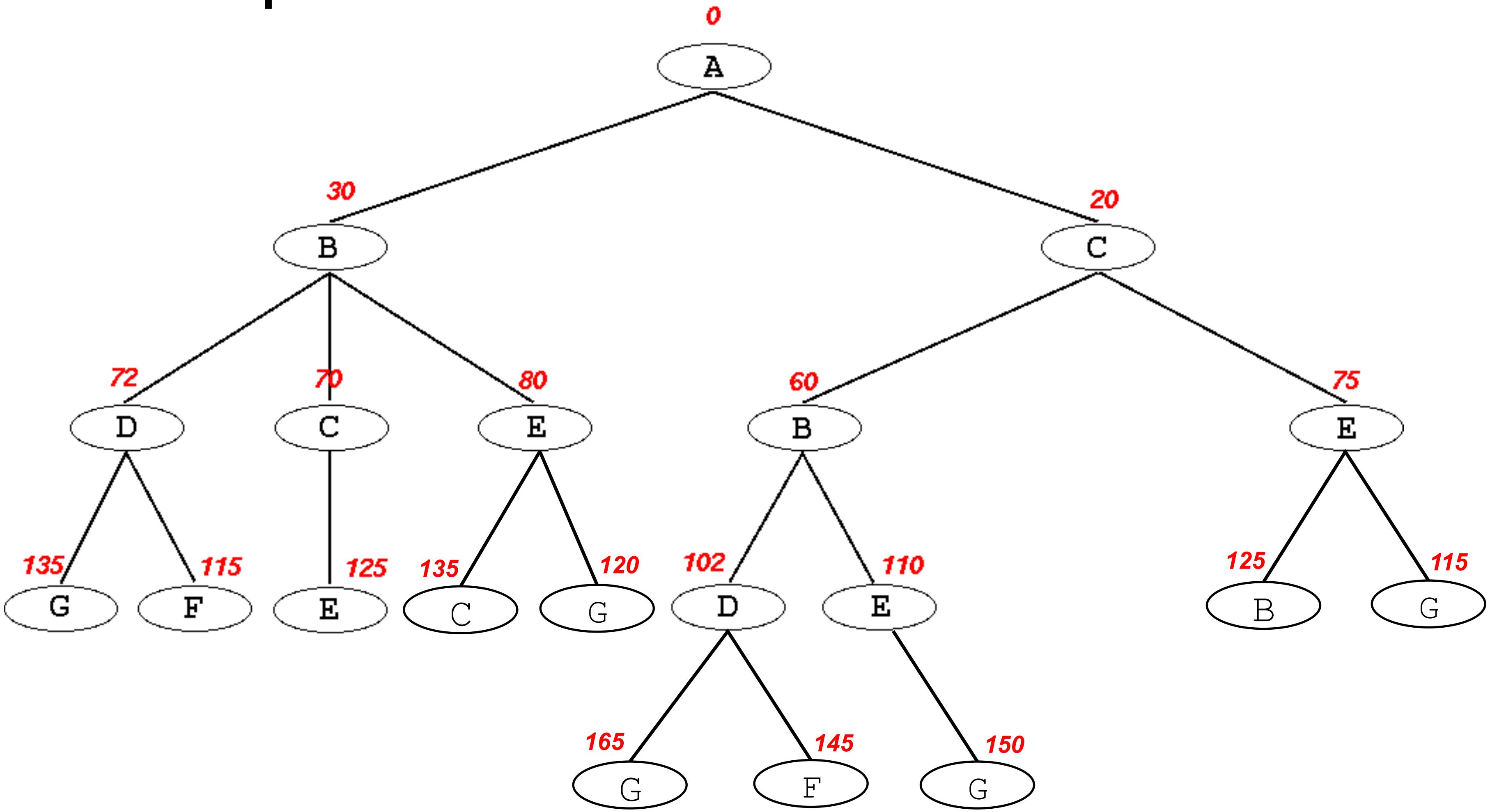
Example: uniform-cost search



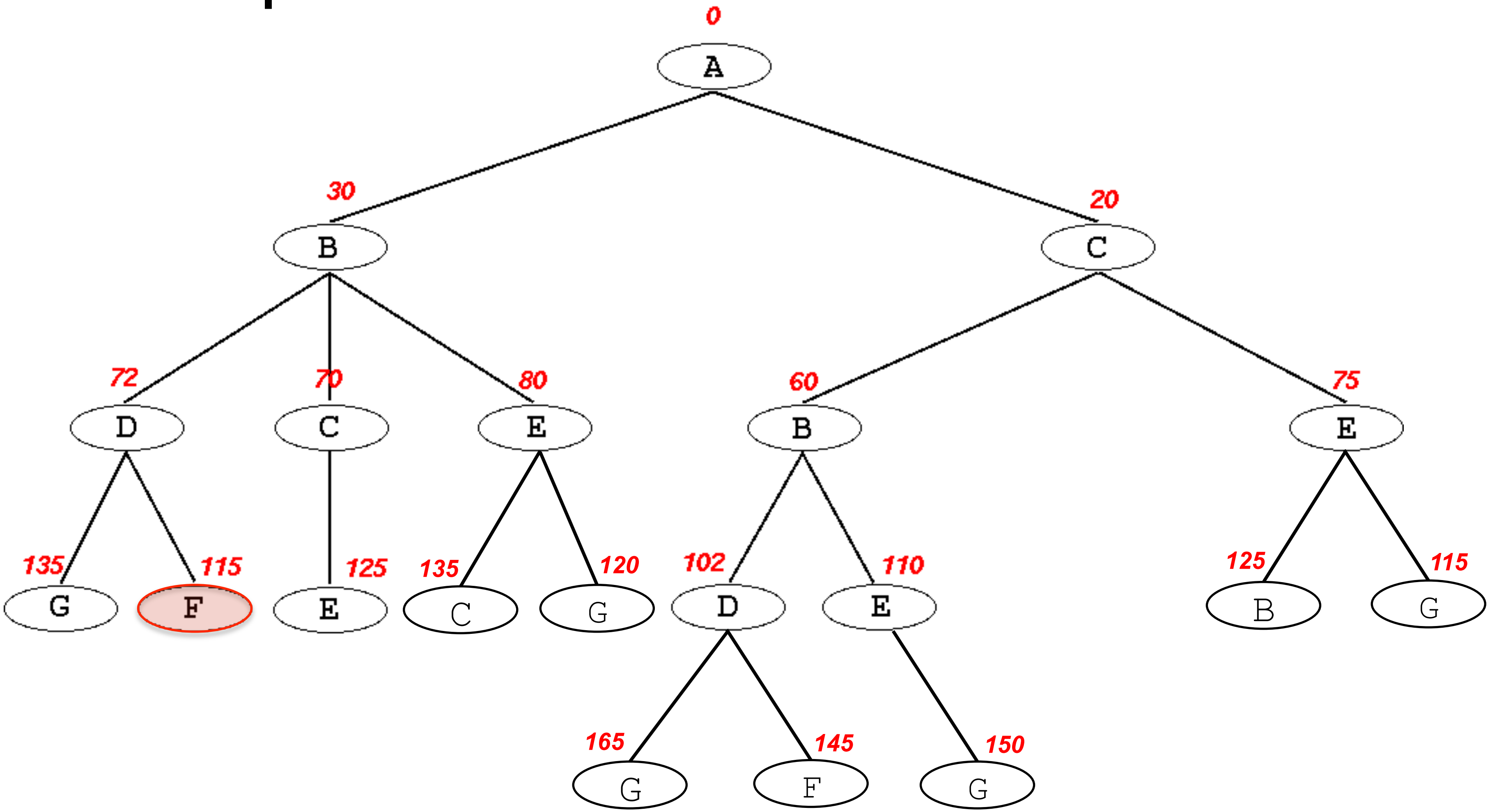
Example: uniform-cost search



Example: uniform-cost search



Example: uniform-cost search



Properties of uniform-cost search

- uniform-cost search is *complete*
- guaranteed to find an *optimal solution* if every operator costs at least $\varepsilon > 0$, i.e, if the cost of a path never decreases
  - if operators can have negative cost an exhaustive search of all nodes is required to find an optimal solution
- time and space complexity is $O(b^{\lceil C^*/\varepsilon \rceil})$ where C^* is the cost of the optimal solution

Uniform-cost search in Prolog

```
% ucs(?initial_state, ?goal_state, ?solution)
% does not include cycle detection
ucs(X,Y,P) :-
    ucs_a(Y,[n(X,[])],R),
    reverse(R,P).

ucs_a(Y,[n(Y,P)|_],P).
ucs_a(Y,[n(S,P1)|Ns],P) :-
    findall(n(S1,[A|P1]),s(S,S1,A),Es),
    append(Ns,Es,O1),
    sort(g_less,O1,O),
    ucs_a(Y,O,P).

% s(?state, ?next_state, ?operator).
s(a,b,go(a,b)).
etc...
```

Computing costs of paths

- uses the generic sorting predicate *sort*/3 defined in lecture 7 (or the *sort*/3 predicate from the SICStus samsort library)
- *g_less*/2 should implement a relation that takes two nodes, *N1* and *N2*, and returns true if the cost of the path represented by *N1* is less than the cost of the path represented by *N2*, i.e., if $g(N1) < g(N2)$
- to compute the cost of a path represented by a node, *g_less*/2 requires auxiliary relations, e.g., for a route planning problem, *g_less*/2 could use the *route_cost*/2 relation from the second lab
- if we need to change the path cost, e.g., if the user asks for a faster rather than a shorter route, we only need to change the first argument to *sort*/3

Computing the costs of paths

- another possibility is store the cost of a path in a node
- in this case:
  - the updated path cost is computed as part of node expansion:
  - *g_less*/2 only has to compare the (already computed) costs of two paths

Computing the costs of paths

```
ucs(X,Y,P) :-
    ucs_b(Y,[n(X,[],0)],R),
    reverse(R,P).

ucs_b(Y,[n(Y,P,_)|_],P).
ucs_b(Y,[n(S,P1,G)|Ns],P) :-
    findall(n(S1,[A|P1],G1),
            (s(S,S1,A,C), G1 is G + C),
            Es),
    append(Ns,Es,O1),
    sort(g_less,O1,O),
    ucs_b(Y,O,P).

% s(?state, ?next_state, ?operator, ?step_cost).
s(a,b,go(a,b),100).
etc...
```

Focusing the search

- using the path cost $g(n)$ allows us to find an optimal solution
- however it does not direct search toward the goal
- in order to focus the search, we need an *evaluation function* which incorporates some *estimate* of the cost of a path *from a state to the closest goal state*

Informed search

- *informed* (or heuristic) search procedures use some form of (often inexact) information to guide the search towards more promising partial solutions

- the *cost* of a partial solution, n, is defined as

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

  where $g(n)$ is the path cost from the start state to n and $h(n)$ is an *estimate* of the cost of going from state n to a goal state

- $h(n)$ is often called the *heuristic function* – the more accurate the heuristic function, the more efficient the search

Informed search procedures

- costs are used to order partial solutions so that the most promising (least cost) nodes are expanded first
  - **greedy search** expands the node with the lowest *h(n)* value, i.e., the node which is estimated to be closest to the goal
  - **A* search** expands the node with the lowest *f(n)* value, i.e., the path through n with the lowest estimated cost
- in contrast, uniform-cost search expands the node with the lowest *g(n)* value, i.e., the node with the lowest path cost

The next lecture

Representing states

Suggested reading:

- Bratko (2001) chapter 12