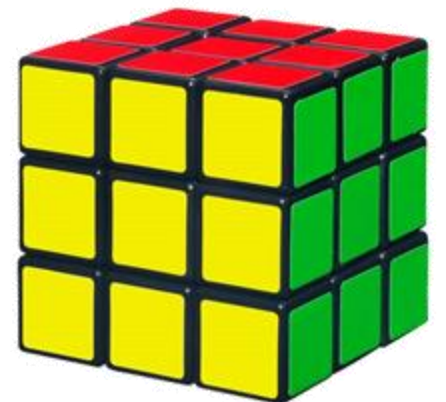


# Search Strategies

28th September 2022

Petter Kristiansen



# Search in State Spaces

- Many problems can be solved by using some form of search in a state space.
- We look at the following methods:
  - Backtracking (Ch. 10)
  - Branch-and-bound (Ch. 10)
  - Iterative deepening (only on these slides, but still part of the curriculum)
  - A\*-search (Ch. 23)

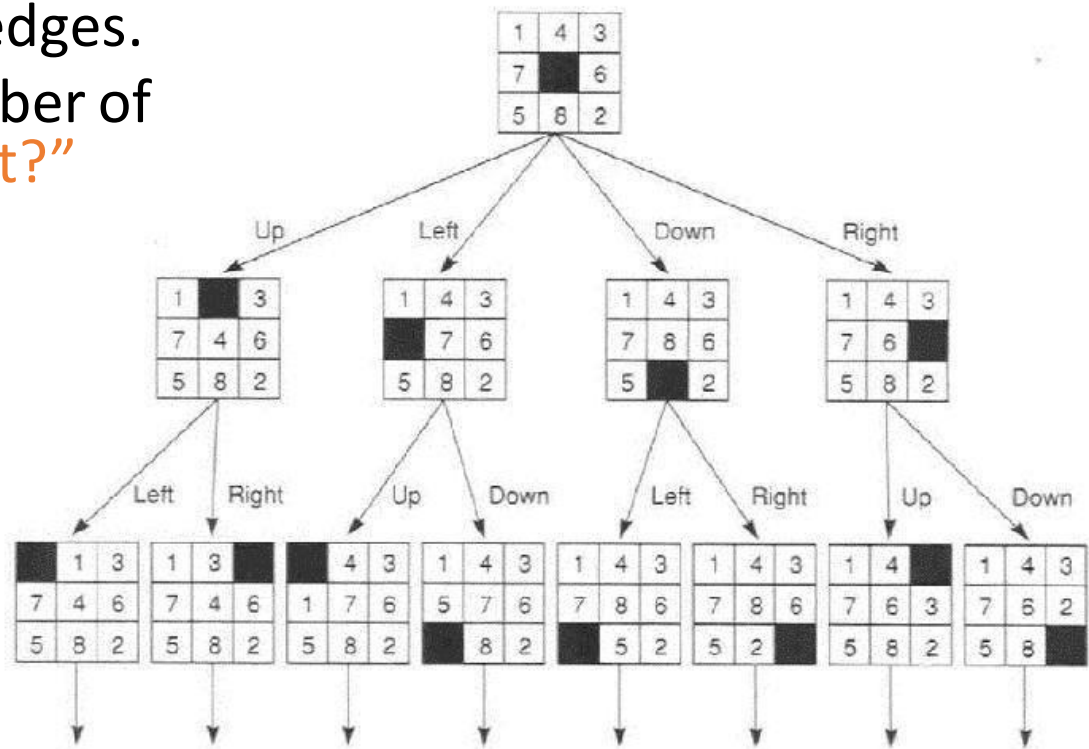
# Search in State Spaces

- *Backtracking* (Ch. 10)
  - Depth-First-Search in a state space: DFS
  - Memory efficient
- *Branch-and-bound* (Ch. 10)
  - Breadth-First-Search: BFS
  - Needs a lot of space: Must store all nodes that have been seen, but not explored further.
  - We can also indicate for each node how «promising» it is (heuristic), and always proceed from the currently most promising one. Natural to use a priority queue to choose next node.
- Iterative deepening (Slides)
  - DFS down to level 1, then to level 2, etc.
  - **Combines**: The memory efficiency of DFS, and the search order of BFS
- Dijkstra's shortest path algorithm (repetition from IN 2010 or similar)
- A\*-search (Ch. 23)
  - Is similar to branch-and-bound, with heuristics and priority queues
  - Can also be seen as an improved version



# State Spaces and Decision Sequences

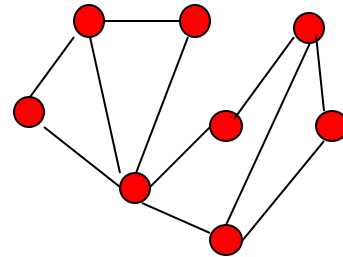
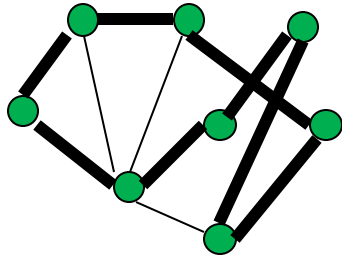
- The *state space* of a system is the set of states the system can be in.
  - Some states are called *goal states*. They are states where we want to end up. No goal state is shown in the figure.
  - Each *search algorithm* will have a way of traversing the states, traversing is usually indicated by directed edges.
  - A search algorithm will usually have a number of decision points: “Where to search / go next?” The full tree with all choices is the *state space tree* for that algorithm.
  - Different algorithms will generate different state space trees.
  - **Main problem:**  
**The state space is usually very large.**



# Models for decision sequences

- There is usually **more than one decision sequence** for a given problem, and they may lead to **different state space trees**.
- **Example:** Find, if possible, a Hamiltonian cycle (see figures below)

Hamiltonian  
Cycle

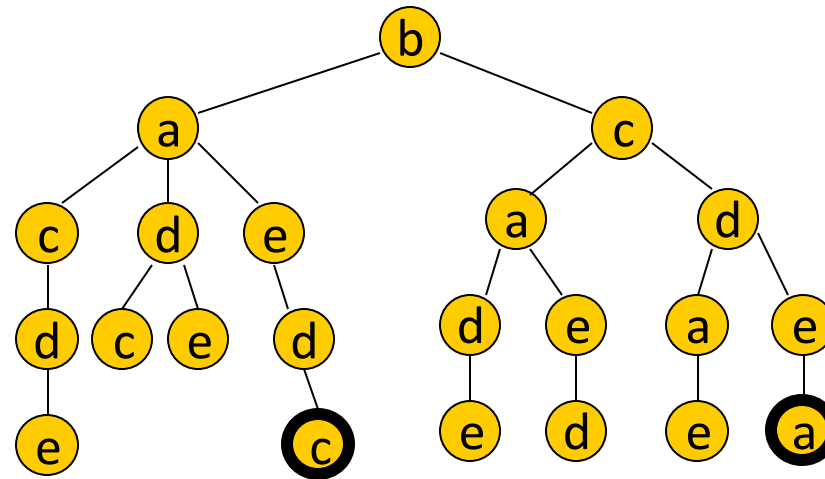
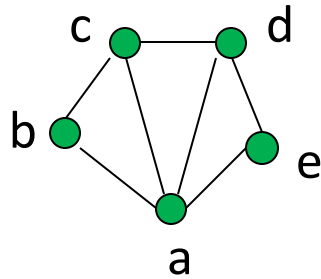


This graph  
obviously has no  
Hamiltonian  
cycle

- There are (at least) two natural decision sequences (and they lead to two different state space trees, as shown on the next slide):
  - Start at any node, and try to grow paths from this node in all possible directions.
  - Start with one edge, and add edges as long as the added edge doesn't form a cycle with already chosen edges (before we have a Hamiltonian cycle).

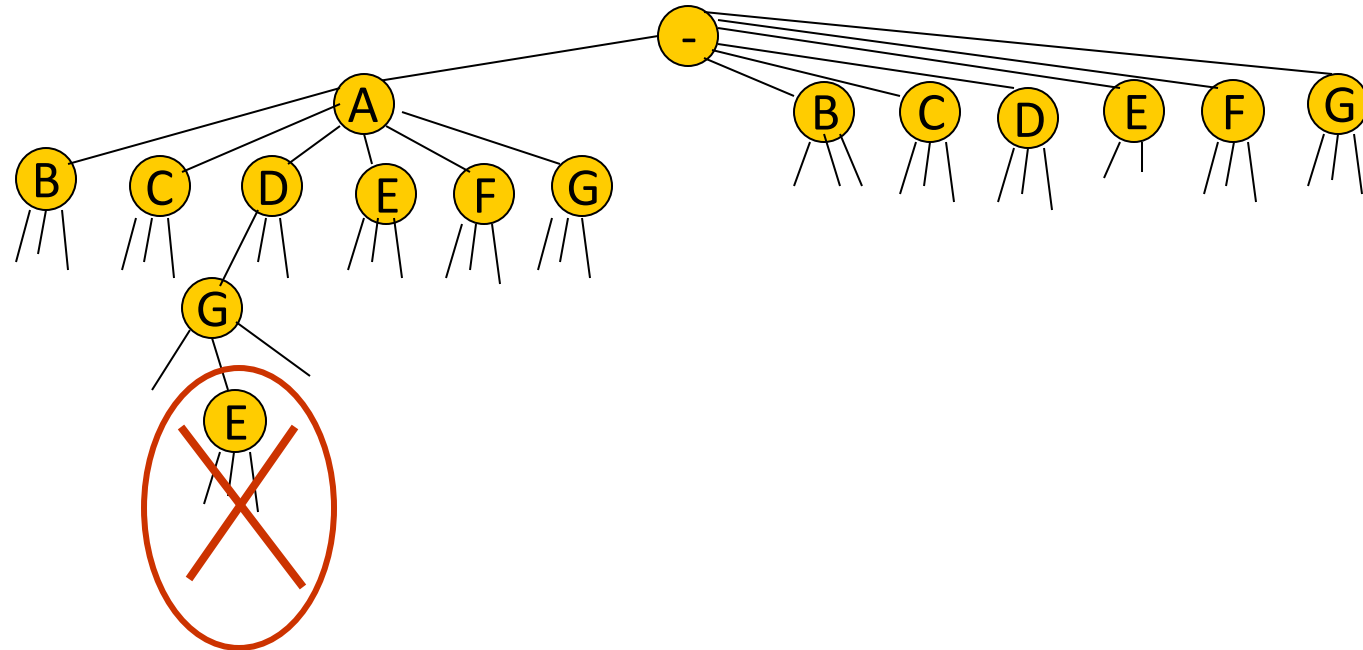
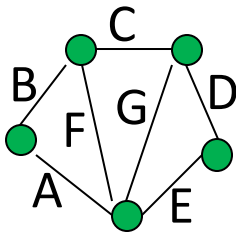
# Models for decision sequences (1)

- A tree structure formed by the first decision sequence:
  - Choose a node and try paths out from from that node.
  - Possible choices in each step: Choose among all unused nodes connected to the current node by an edge.

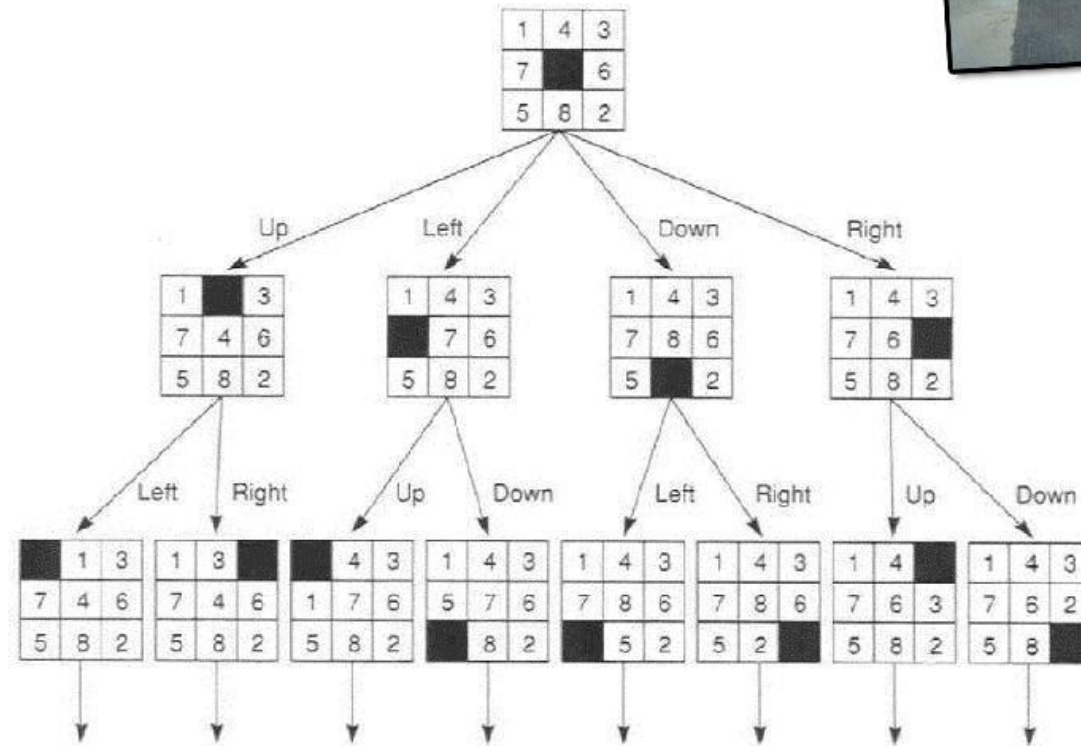


# Models for decision sequences (2)

- A tree structure formed by the second model:
  - Start with one edge, and add edges as long as the added edge doesn't form a cycle with already chosen edges (before we have a Hamiltonian cycle).

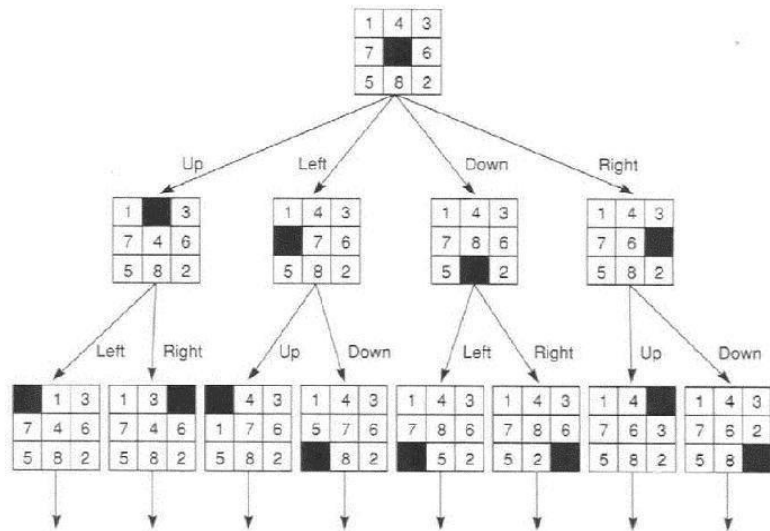


# State spaces and decision sequences



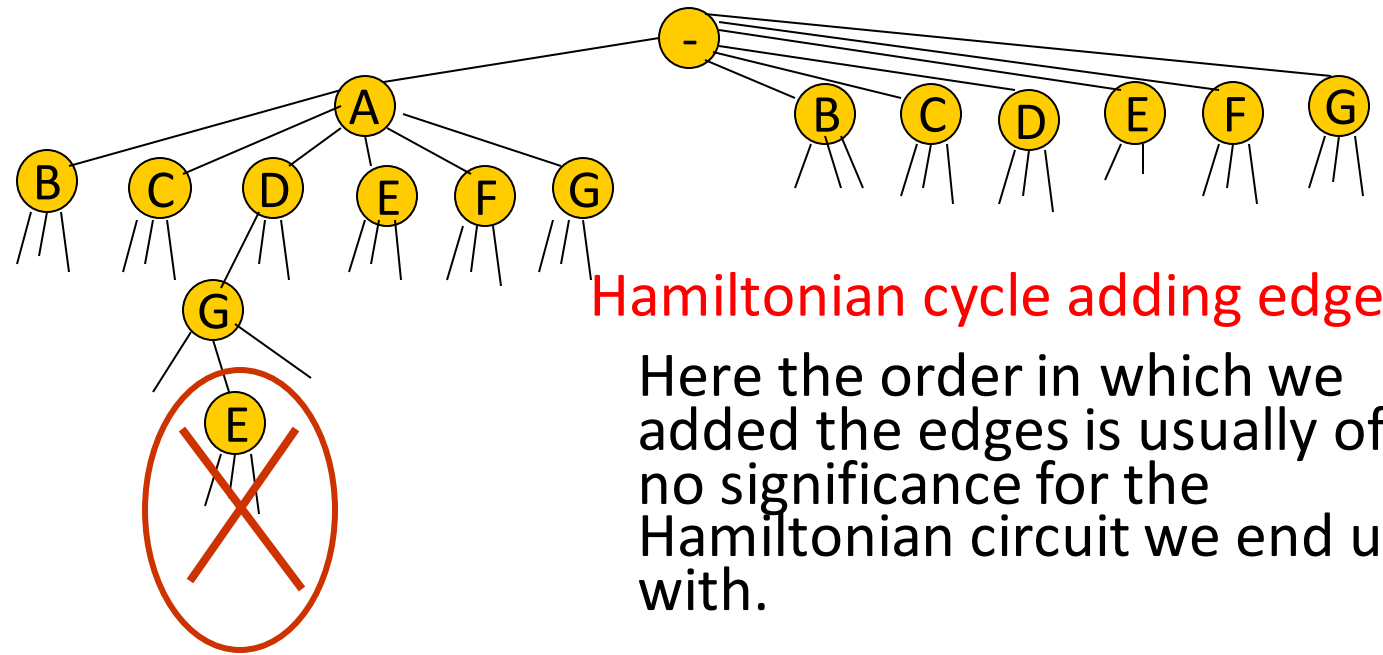
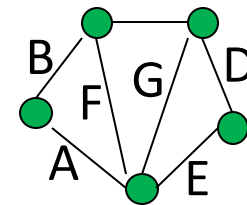


# Sometimes the path leading to the goal node is as important part of the solution



8-puzzle:

Here the path leading to the goal node is the sequence of moves we should perform to solve the puzzle



Hamiltonian cycle adding edges:

Here the order in which we added the edges is usually of no significance for the Hamiltonian circuit we end up with.

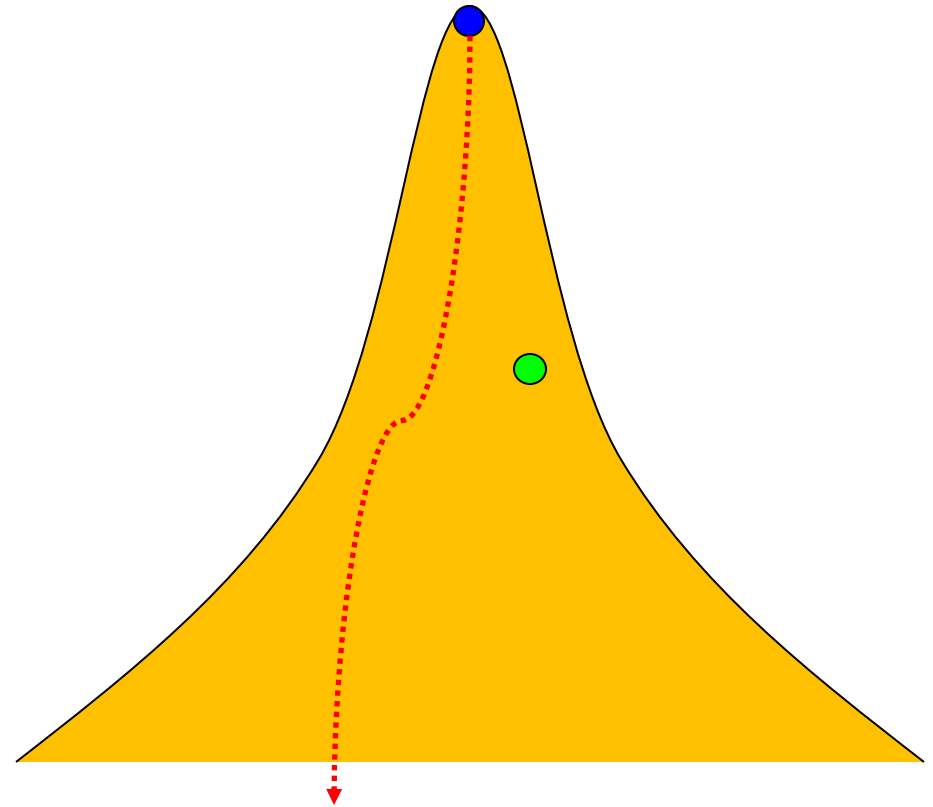
# Backtracking and Depth-First-Search

# Backtracking and Depth-First-Search

A template for implementing depth-first-search may look like this:

```
procedure DFS(v)
{
  IF <v is a goal node> THEN return ``...``
  v.visited = TRUE;
  FOR <each neighbour w of v> DO
    IF not w.visited THEN DFS(w) FI
  OD
}
```

It can not only be used for trees, but also for graphs, because of this



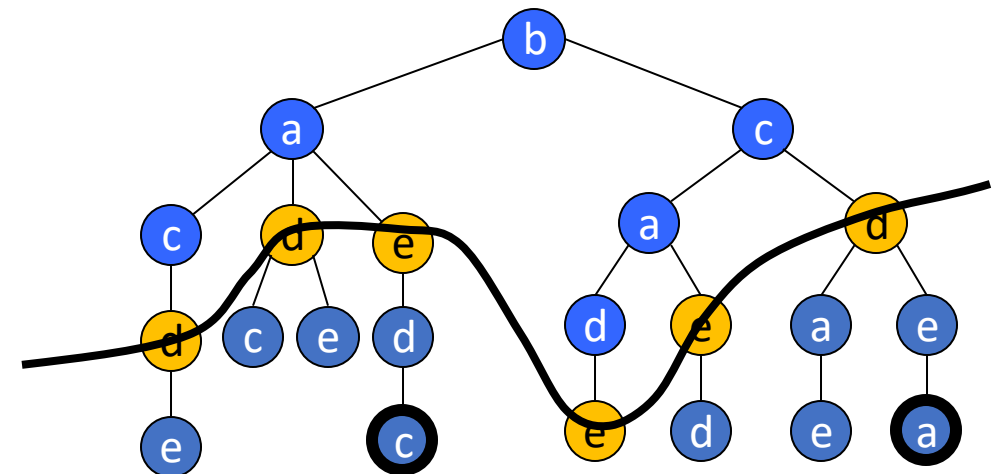
# Backtracking and Depth-first-search (DFS)

- Searches the state space tree *depth first with backtracking*, until it reaches a goal state (or has visited all states).
- The easiest implementation is usually to use a recursive procedure.
- Memory efficient – only « $O(\text{the depth of the tree})$ ».
- If the edges have lengths and we e.g. want a shortest possible Hamiltonian cycle, we can use heuristics to choose the most promising direction first (e.g. choose the shortest legal edge from where you are now).
- One has to use *pruning* (or *bounding*) as often as possible.  
**An exhaustive search usually requires exponential time!**
  - **Main pruning principle: Don't enter subtrees that cannot contain a goal node.**  
(The difficulty is to find where this is the case.)

Branch-and-bound / Breadth-  
First-Search (BFS)

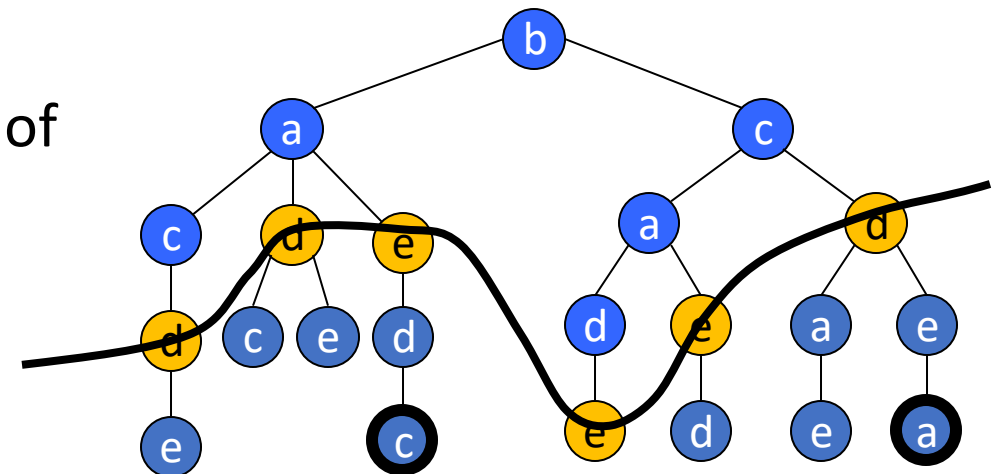
# Branch-and-bound / Breadth-First-Search (BFS)

- Uses some form of breadth-first-search.
- We have three sets of nodes:
  1. The *finished nodes* (dark blue). Often do not need to be stored.
  2. The *live nodes* (orange) seen but not explored further. Large set, that must be stored.
  3. The *unseen nodes* (light blue). We often don't have to look at all of them.
- The *live nodes* (orange) will always be a cut through the state-space tree (or likewise if it is a graph)
- The main step:
  - Choose a node  $n$  from the set of *live nodes* (LiveNodes).
  - If  $N$  is a goal node, then we are finished, **ELSE**:
  - Take  $n$  out of the *LiveNodes set* and insert it into the *finished nodes*.
  - Insert all children of  $N$  into the *LiveNodes set*.
  - If we are searching a graph, only insert *unseen ones*.



# Branch-and-bound / Breadth-First-Search (BFS)

- Three strategies:
  - The **LiveNodes set** is a FIFO-queue
    - We get traditional breadth first
  - The **LiveNodes set** is a LIFO-queue
    - The order will be similar to depth-first, but not exactly
  - The **LiveNodes set** is a priority queue,
    - We can call this priority search
    - If the priority stems from a certain kind of heuristics, then this will be A\*-search (slides below)



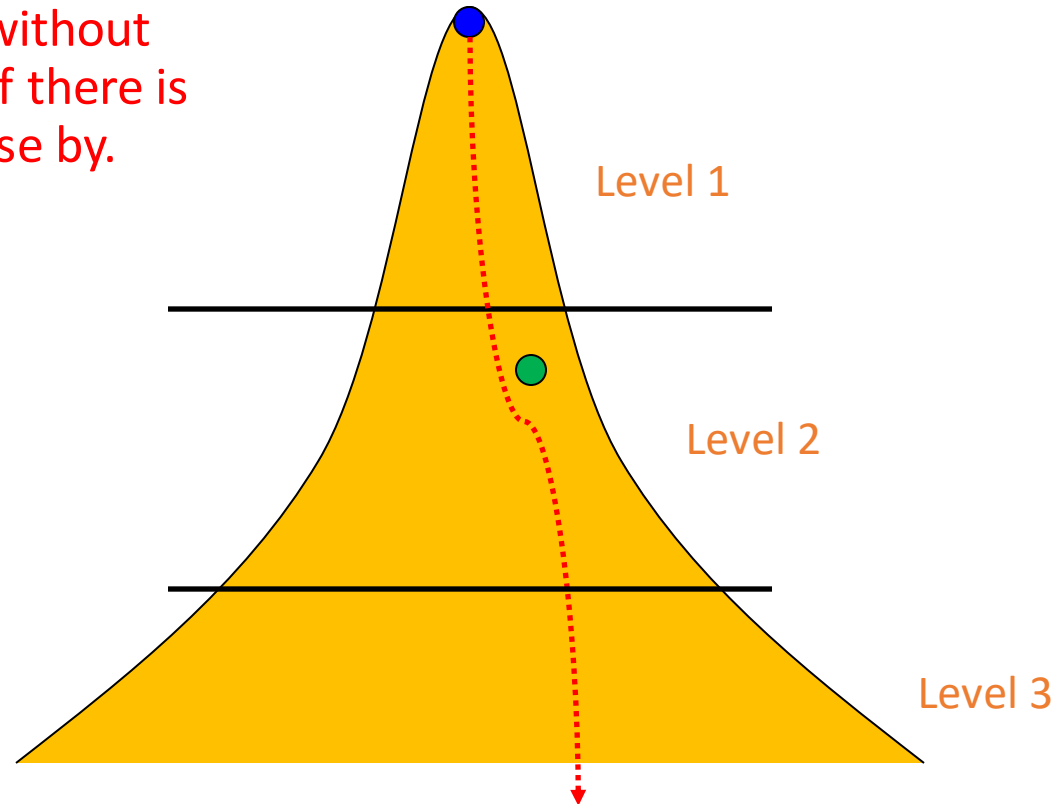
Iterative deepening



# Iterative deepening

Not in the textbook, but part of the curriculum

A drawback with DFS is that you can end up going very deep down into one branch without finding anything, even if there is a shallow goal node close by.



- We can avoid this by first doing DFS to level one, then to level two, etc.
- With a reasonable branching factor, this will not be too much extra work, and we are always memory efficient.
- We only test for goal nodes at levels we have not been on before.

# Assignment (iterative deepening)

Adjust the DFS program to do iterative deepening:

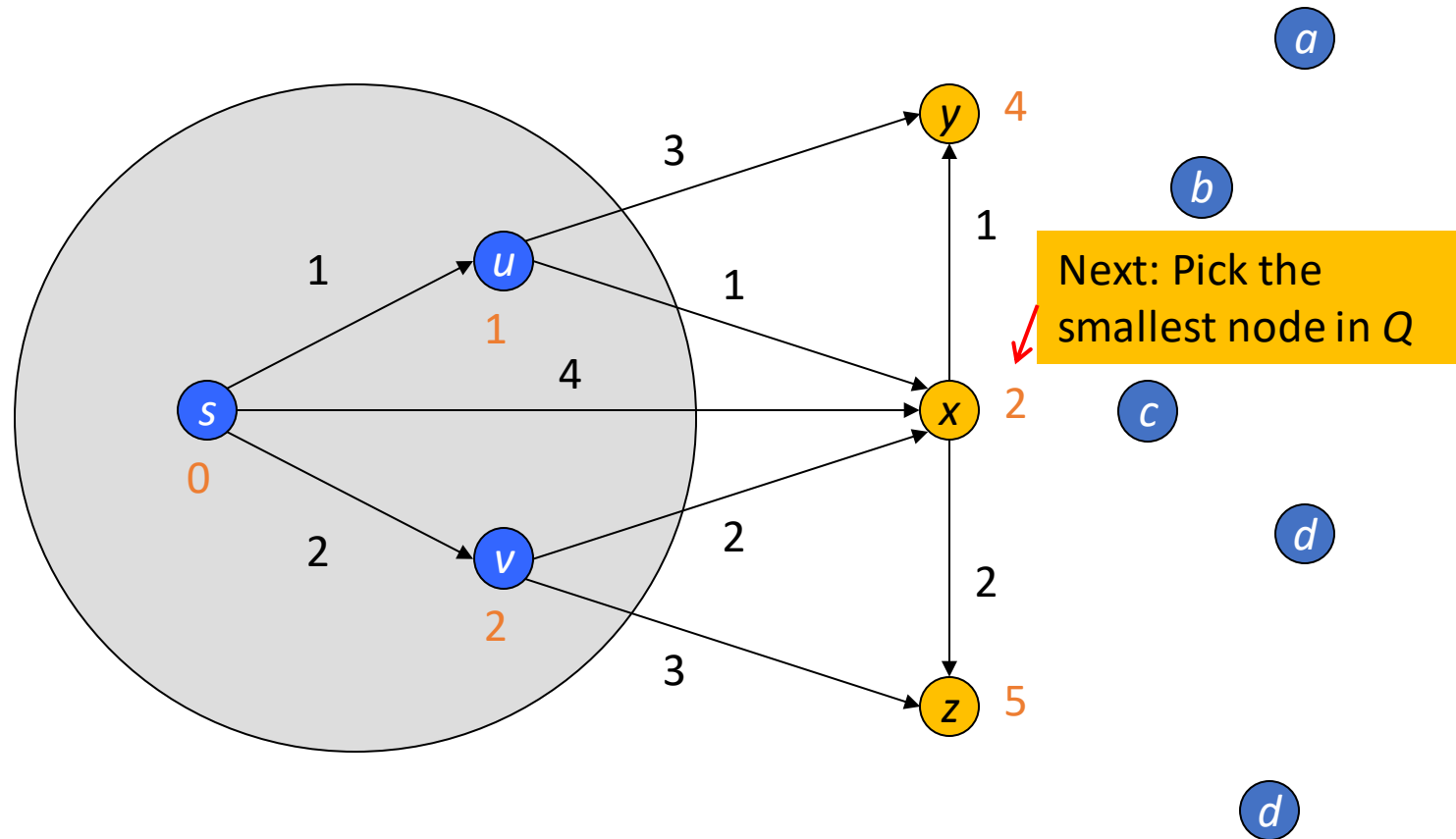
```
procedure DFS(v)
{
  IF <v is a goal node> THEN return ``...`` FI
  v.visited=TRUE
  FOR <each neighbor w of v> DO
    IF not w.visited THEN DFS(w) FI
  OD
}
```

We assume that the test for deciding whether a given node is a goal node is expensive, and we shall therefore only test this for the "new levels" (only once for each node).

Discuss how iterative deepening will work for a directed graph.

Dijkstra's algorithm

# Dijkstra's algorithm for single source shortest paths in directed graphs (Ch. 23)



«Tree nodes»  
(The finished nodes)

$Q$ : The priority queue  
(The «live nodes»)

Unseen nodes

# Dijkstra's algorithm

```
procedure Dijkstra(graph G, node source)
  for each each node v in G do
    v.dist := ∞
    v.previous := NIL
  od
  source.dist := 0
  Q := { source }
  while Q is not empty do
    u := extract_min(Q)
    for each neighbor v of u do
      x = length(u, v) + u.dist
      if x < v.dist then
        v.dist := x
        v.previous := u
      fi
    od
  od
end
```

// Initialization  
// Marks all as unseen nodes  
// Pointer to remember the path back to source  
// Distance from source to itself  
// The initial priority queue only contains source  
// Node in Q closest to source. Is removed from Q  
// Key in priority queue is distance from source  
// Nodes in the "tree" will never pass this test  
// Shortest path "back towards the source"

Could already here discard nodes in the tree

A- / A\*-search

# A-/A\*-search (Hart, Nilsson, Raphael, 1968)

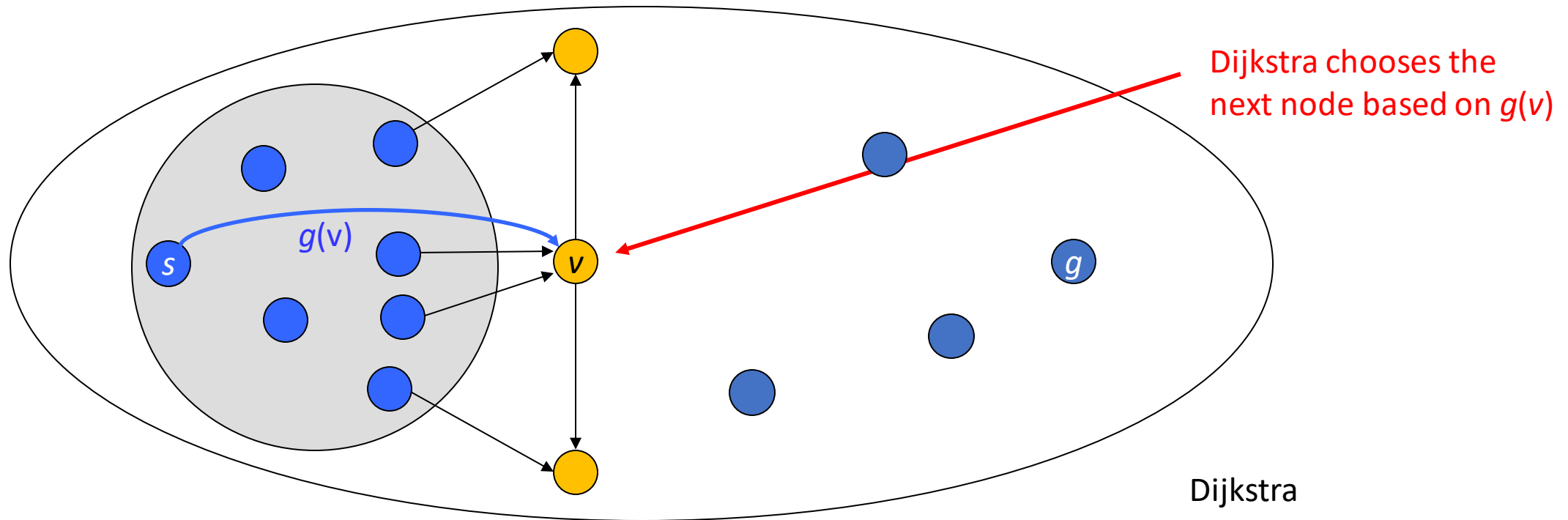
- Backtracking / depth-first, LIFO / FIFO, branch-and-bound, breadth-first and Dijkstra's algorithm only use **local information** when choosing the next step
- A\*-search is similar to Dijkstra's algorithm, but it uses a global heuristic (a "qualified guess") to make better choices from Q in each step
- Widely used in AI and knowledge based systems
- A\*-search (like Dijkstra's alg.) is useful for problems where we have
  - An explicit or implicit graph of "states"
  - There is a *start state* and a number of *goal states*
  - The (directed) edges represent legal state transitions, and they all have a cost

And (like with Dijkstra's alg.) the aim is to find the cheapest (shortest) path from the start node to a goal node

- A\*-search: If we for each node in Q can "guess" how far it is to a goal node, then we can often speed up the algorithm considerably!

# A-search – heuristic

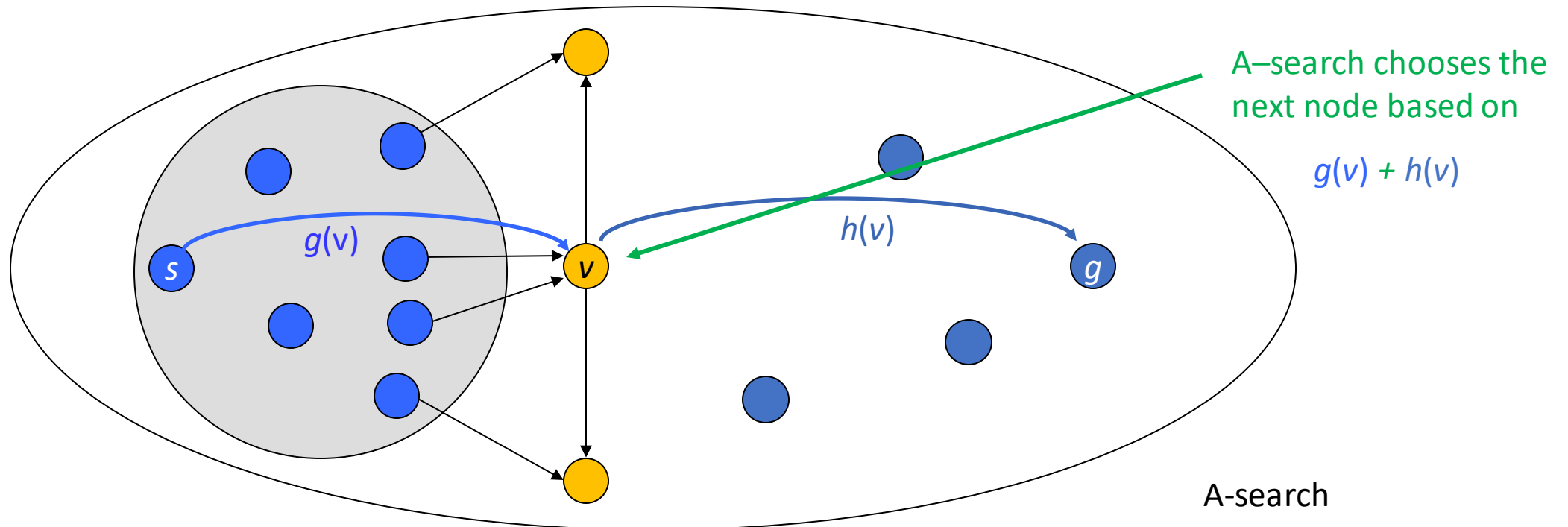
- The strategy is a sort of breadth-first search, like Dijkstra's algorithm
  - However, we now use an estimate  $h(v)$  for the shortest path from the node  $v$  to some goal node  $g$  ( $h$  for *heuristic*)
  - The value we use for choosing the best next node is now  $f(v) = g(v) + h(v)$
  - Thus, we get a priority-first search with this value as priority





# A-search – heuristic

- The strategy is a sort of breadth-first search, like Dijkstra's algorithm
  - However, we now use an estimate  $h(v)$  for the shortest path from the node  $v$  to some goal node  $g$  ( $h$  for *heuristic*)
  - The value we use for choosing the best next node is now  $f(v) = g(v) + h(v)$
  - Thus, we get a priority-first search with this value as priority



# A-search and types of heuristics

The function  $h(v)$  should be an estimate of the distance from  $v$  to the closest goal node:

- No limitations on the heuristic  $h(v)$
- $h(v)$  is never larger than the actual shortest path to the closest goal node (**underestimate**)
- $h(v)$  is monotone (explained on slides below)

# A-search and inadmissible heuristics

- With no limitations on the heuristic  $h(v)$ 
  - We may get an incorrect answer

Why?

What kind kind of “algorithm” do we get?

# A-search and admissible heuristics

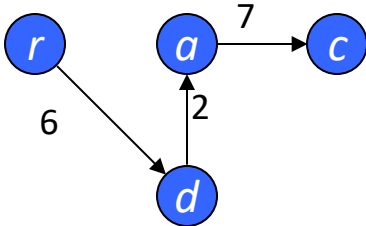
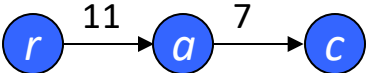
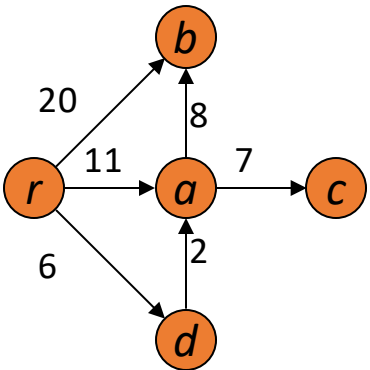
- If  $h(v)$  is never larger than the actual shortest path to the closest goal node,
  - we will eventually get the correct result
- However, we will often have to go back to nodes that we “thought we were finished with”  
Nodes in the tree will have to go back into the priority queue, when/if we find a shorter path to them than the one we have calculated already
  - This can result in a lot of extra work
- Dijkstra’s algorithm never moves a tree-node back into the queue, and still gets the correct answer. We will put requirements on  $h(v)$  so that this will also be true for the A-search algorithm above.

# Exercise (admissible heuristic)

- Study the example in figure 23.5 (the textbook, page 723). It demonstrates some of what is said on the previous slide:

If you do not move a node back into Q from the tree (when a shorter path to the node is found), you may not find the shortest path to the goal

- The drawings below is from that example:



<i>v</i>	<i>r</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
<i>h(v)</i>	-	23	20	15	29

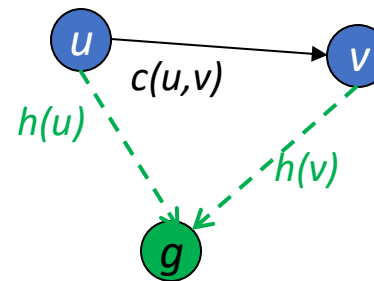
# A\*-search and monotone heuristics

- If  $h(v)$  is never larger than the shortest distance to a goal, we know that the search will terminate, but maybe slowly
- If we restrict  $h(v)$  further, we get more efficient algorithms:
  1. The function  $h(v)$  is never larger than the actual shortest distance to the closest goal-node
  2.  $h(g) = 0$  for all goal nodes  $g$
  3. And a sort of “*triangle inequality*” must hold:

If there is a transition from node  $u$  to node  $v$  with cost  $c(u,v)$ , then the following should hold:

$$h(u) \leq c(u,v) + h(v)$$

In this case,  $h(v)$  is said to be **MONOTONE**



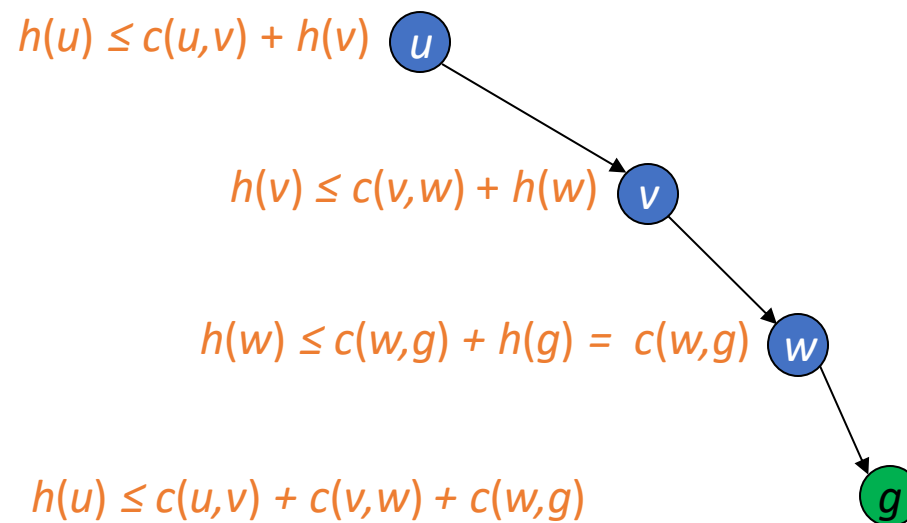
# A\*-search and monotone heuristics

- (From the previous slide) Requirement 2 and 3 for *monotonicity* on  $h$ :
  2. Every goal node  $g$  must have  $h(g) = 0$
  3. If there is an edge from  $v$  to  $w$  with weight  $c(v,w)$ , then we must always have  $h(v) \leq c(v,w) + h(w)$
- A nice thing here is that if these two requirements are fulfilled:
  - Then Requirement 1 (that  $h(v)$  is never larger than the actual shortest distance to a goal) is automatically fulfilled

## Sketch of a proof:

We assume that  $u \rightarrow v \rightarrow w \rightarrow g$  is a shortest path from  $u$  to a goal state  $g$ .

We set up the inequalities we know:



If we combine the above, we get:  $h(u) \leq c(u,v) + c(v,w) + c(w,g)$

# An aside on naming

- Without any limitations on the heuristic the algorithm is not guaranteed to find the right answer (It is not even really an algorithm!)
- A heuristic that always is an *underestimate* ( $\leq$ ) is said to be *admissible* (it leads to an algorithm that actually solves all instances of the problem)
- With an admissible heuristic we get A-search. This algorithm finds the correct answer, but may be sub-optimal with respect to speed. (The original algorithm was called A1)
- A heuristic is said to be *monotone* if
  - $h(g) = 0$  for all goal nodes  $g$ , and
  - $h(u) \leq c(u,v) + h(v)$  for all  $u$  and  $v$  with associated transition cost (edge with weight)  $c(u,v)$
- With a monotone heuristic we get **A\*-search**. (It was proved to be optimal, and improves previous algorithms called A1 and A2)

(Not all sources are too strict with this naming, many use the name A\*-search for all types of admissible heuristics.)



# Relation to Dijkstra's algorithm

- If we use  $h(v) = 0$  for all nodes,  $A^*$  becomes Dijkstra's algorithm
- With a better heuristic we hope to work *less* with paths that do not lead to solutions, so that the algorithm will run faster
- With a heuristic we will likely remove nodes from  $Q$  in a different order than with Dijkstra's algorithm
- Thus, initially we can no longer be certain that when  $v$  is moved from  $Q$  to the tree, it has the correct shortest path length  $v(g)$ .

BUT luckily, we can prove this (proposition 23.3.2 in the book):

- If  $h$  is monotone, then values of  $g(v)$  and  $parent(v)$  will always be correct when  $v$  is removed from  $Q$  to become a tree node.
- Therefore, we never need to go back to tree nodes, move them back into  $Q$ , and update their variables.

There is a misprint at page 724, in formula 23.3.7:

Where: ...  $h(v) + h(v)$  ... appears, it should instead be: ...  $h(v) \leq g(v) + h(v)$  ...

# A\*-search – the data for the algorithm

- We have a directed graph  $G$  with edge weights  $c(u, v)$ , a start node  $s$ , a number of goal-nodes, and a monotone heuristic function  $h(u)$ , (often set in all nodes during initialization, and never changed).
- In addition, each node  $v$  has the following variables:
  - $g$ : This variable will normally change many times during the execution of the algorithm, but its final value (after being moved to the tree) will be the length of the shortest path from the start node to  $v$ .
  - $parent$ : This variable will end up pointing to the parent in a tree of shortest paths back to the start node.
  - $f$ : This variable will all the time have the value  $g(v) + h(v)$ , that is, an estimate of the path length from the start node to a goal node, through the node  $v$ .
- We keep a priority queue  $Q$  of nodes, where the value of  $f$  is used as priority
  - This queue is initialized to contain only the start node  $s$ , with  $g(s) = 0$ .  
(This initialization is missing in the description of the algorithm at page 725.)
  - The value of  $f$  will change during the algorithm, and the node must then be «moved» in the priority queue.
- The nodes that is *not* in  $Q$  at the moment are partitioned into two types:
  - Tree nodes: In these the parent pointer is part of a tree with the start node as root. These nodes have all been in  $Q$  earlier.
  - Unseen nodes (those that we have not touched up until now).

# A\*-search – the algorithm

Repeat until Q is empty: Find the node  $v$  in Q with the smallest f-value

1. If  $v$  is a goal node, the algorithm should terminate, and  $g(u)$  and  $parent(u)$  indicates the shortest path (backwards) from the start node to  $v$ .
2. Otherwise, remove  $v$  from Q, and let it become a tree node (it now has its parent pointer and  $g(v)$  set to correct values)
  - Look at all the unseen neighbors  $w$  of  $v$ , and for each do the following:
    - Set  $g(w) = c(v,w) + g(v)$
    - Set  $f(w) = g(w) + h(w)$
    - Set  $parent(w) = v$
    - Insert  $w$  into Q
  - Look at all neighbours  $w'$  of  $v$  that are in Q, and for each of them do:
    - If  $g(w') > g(v) + c(v, w')$  then
      - set  $g(w) = c(v,w) + g(v)$
      - set  $parent(w)=v$

Note that we (as in Dijkstra's algorithm ) do *not* look at neighbours of  $v$  that are tree nodes. That this will work needs a proof, see next slides

# Proof that A\*-search works (proposition 23.3.2)

Understanding the induction is part of the curriculum, but not the details (book does not use induction)

**Induction hypothesis:** Proposition 23.3.2 is true for all nodes  $u$  that are moved from  $Q$  into the tree before  $v$ . That is,  $g(u)$  and  $parent(u)$  is correct for all such  $u$

**Induction step:** We have to show that after  $v$  is moved to the tree, this is also true for the node  $v$  (and none of the old tree nodes are changed)

**Definition:** Let generally  $g^*(u)$  be the length of the (actual) shortest path from the start node to a node  $u$

We want to show that  $g(v) = g^*(v)$  after  $v$  is moved from  $Q$  to the tree

We examine the situation when  $v$  is moved from  $Q$  to the tree, and look at the node sequence  $P$  from the start node  $s$  to  $v$ , following the parent pointers from  $v$

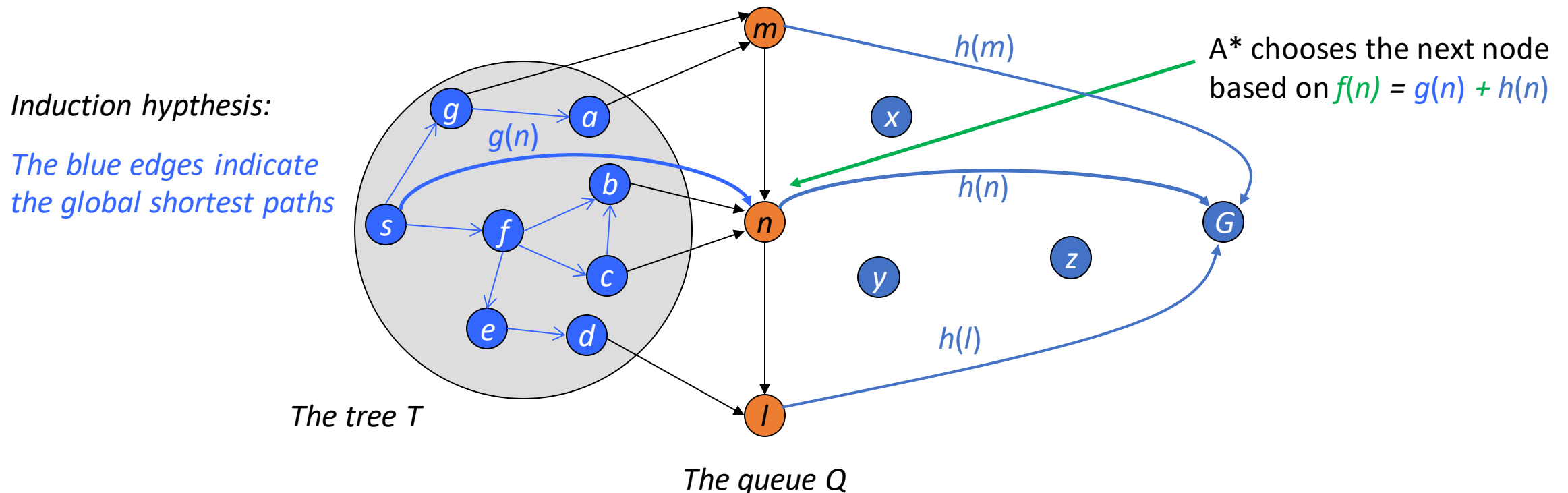
$$P: s = v_0, v_1, v_2, \dots, v_k, v_{k+1}, \dots, v_j = v$$

We now *assume* that  $v_0, v_1, \dots, v_k$ , but not  $v_{k+1}$ , where  $k+1 < j$ , have become tree nodes when  $v$  is removed from  $Q$ , so that  $v_{k+1}$  is in  $Q$  when  $v$  is removed from  $Q$

**We shall show that this cannot be the case when  $h$  is monotone**

# Illustrating the proof for A\*-search

- The value we use for choosing the best next node  $n$  is now  $f(n) = g(n) + h(n)$
- NB: We assume that the heuristic function  $h$  is monotone
- The important point is to show that there cannot be a shorter path to  $n$  that pass through another node outside the tree, e.g. through  $m$ .



# End of proof that $A^*$ works

From the monotonicity of  $h$  we know that in our sequence  $P$  (for  $i = 0, 1, \dots, j-1$ )

$$g^*(v_i) + h(v_i) \leq g^*(v_i) + c(v_i, v_{i+1}) + h(v_{i+1})$$

Since the edge  $(v_i, v_{i+1})$  is part of the shortest path to  $v_{i+1}$ , we have:

$$g^*(v_{i+1}) = g^*(v_i) + c(v_i, v_{i+1})$$

From these two together, we get:

$$g^*(v_i) + h(v_i) \leq g^*(v_{i+1}) + h(v_{i+1})$$

By letting  $i$  be  $k+1, k+2, \dots, j-1$  respectively, we get

$$g^*(v_{k+1}) + h(v_{k+1}) \leq g^*(v_j) + h(v_j) = g^*(v) + h(v)$$

From the induction hypotheses we know that  $g(v_k) = g^*(v_k)$ , and (by looking at the actions done when  $v_k$  was taken out of  $Q$ ) that  $g(v_{k+1}) = g^*(v_{k+1})$ , even if it occurs in  $Q$  (the heuristic does not interfere with that).

So we know:


$$f(v_{k+1}) = g(v_{k+1}) + h(v_{k+1}) = g^*(v_{k+1}) + h(v_{k+1}) \leq g^*(v) + h(v) \leq g(v) + h(v) = f(v)$$

Here, all ' $\leq$ ' must be equalities, otherwise  $f(v_{k+1}) < f(v)$ , and then  $v$  would not have been taken out of  $Q$  before  $v_{k+1}$ . Therefore  $g^*(v) + h(v) = g(v) + h(v)$  and thus

$$g^*(v) = g(v)$$

# End of proof that $A^*$ works

From the monotonicity of  $h$  we know that in our sequence  $P$  (for  $i = 0, 1, \dots, j-1$ )

$$g^*(v_i) + h(v_i) \leq g^*(v_i) + c(v_i, v_{i+1}) + h(v_{i+1})$$


Since the edge  $(v_i, v_{i+1})$  is part of the shortest path to  $v_{i+1}$ , we have:

$$g^*(v_{i+1}) = g^*(v_i) + c(v_i, v_{i+1})$$

From these two together, we get:

$$g^*(v_i) + h(v_i) \leq g^*(v_{i+1}) + h(v_{i+1})$$

By letting  $i$  be  $k+1, k+2, \dots, j-1$  respectively, we get

$$g^*(v_{k+1}) + h(v_{k+1}) \leq g^*(v_j) + h(v_j) = g^*(v) + h(v)$$

From the induction hypotheses we know that  $g(v_k) = g^*(v_k)$ , and (by looking at the actions done when  $v_k$  was taken out of  $Q$ ) that  $g(v_{k+1}) = g^*(v_{k+1})$ , even if it occurs in  $Q$  (the heuristic does not interfere with that).

So we know:

$$f(v_{k+1}) = g(v_{k+1}) + h(v_{k+1}) = g^*(v_{k+1}) + h(v_{k+1}) \leq g^*(v) + h(v) \leq g(v) + h(v) = f(v)$$

Here, all ' $\leq$ ' must be equalities, otherwise  $f(v_{k+1}) < f(v)$ , and then  $v$  would not have been taken out of  $Q$  before  $v_{k+1}$ . Therefore  $g^*(v) + h(v) = g(v) + h(v)$  and thus

$$g^*(v) = g(v)$$

# End of proof that $A^*$ works

From the monotonicity of  $h$  we know that in our sequence  $P$  (for  $i = 0, 1, \dots, j-1$ )

$$g^*(v_i) + h(v_i) \leq g^*(v_i) + c(v_i, v_{i+1}) + h(v_{i+1})$$


Since the edge  $(v_i, v_{i+1})$  is part of the shortest path to  $v_{i+1}$ , we have:

$$g^*(v_{i+1}) = g^*(v_i) + c(v_i, v_{i+1})$$


From these two together, we get:

$$g^*(v_i) + h(v_i) \leq g^*(v_{i+1}) + h(v_{i+1})$$

By letting  $i$  be  $k+1, k+2, \dots, j-1$  respectively, we get

$$g^*(v_{k+1}) + h(v_{k+1}) \leq g^*(v_j) + h(v_j) = g^*(v) + h(v)$$

From the induction hypotheses we know that  $g(v_k) = g^*(v_k)$ , and (by looking at the actions done when  $v_k$  was taken out of  $Q$ ) that  $g(v_{k+1}) = g^*(v_{k+1})$ , even if it occurs in  $Q$  (the heuristic does not interfere with that).

So we know:

$$f(v_{k+1}) = g(v_{k+1}) + h(v_{k+1}) = g^*(v_{k+1}) + h(v_{k+1}) \leq g^*(v) + h(v) \leq g(v) + h(v) = f(v)$$

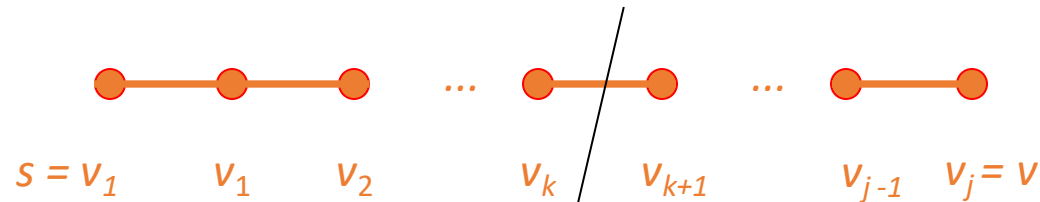
Here, all ' $\leq$ ' must be equalities, otherwise  $f(v_{k+1}) < f(v)$ , and then  $v$  would not have been taken out of  $Q$  before  $v_{k+1}$ . Therefore  $g^*(v) + h(v) = g(v) + h(v)$  and thus

$$g^*(v) = g(v)$$



# End of proof that A\* works

A sequence  $P$  of nodes from  $s$  to  $v$ , following parent pointers.  $h$  is monotone.

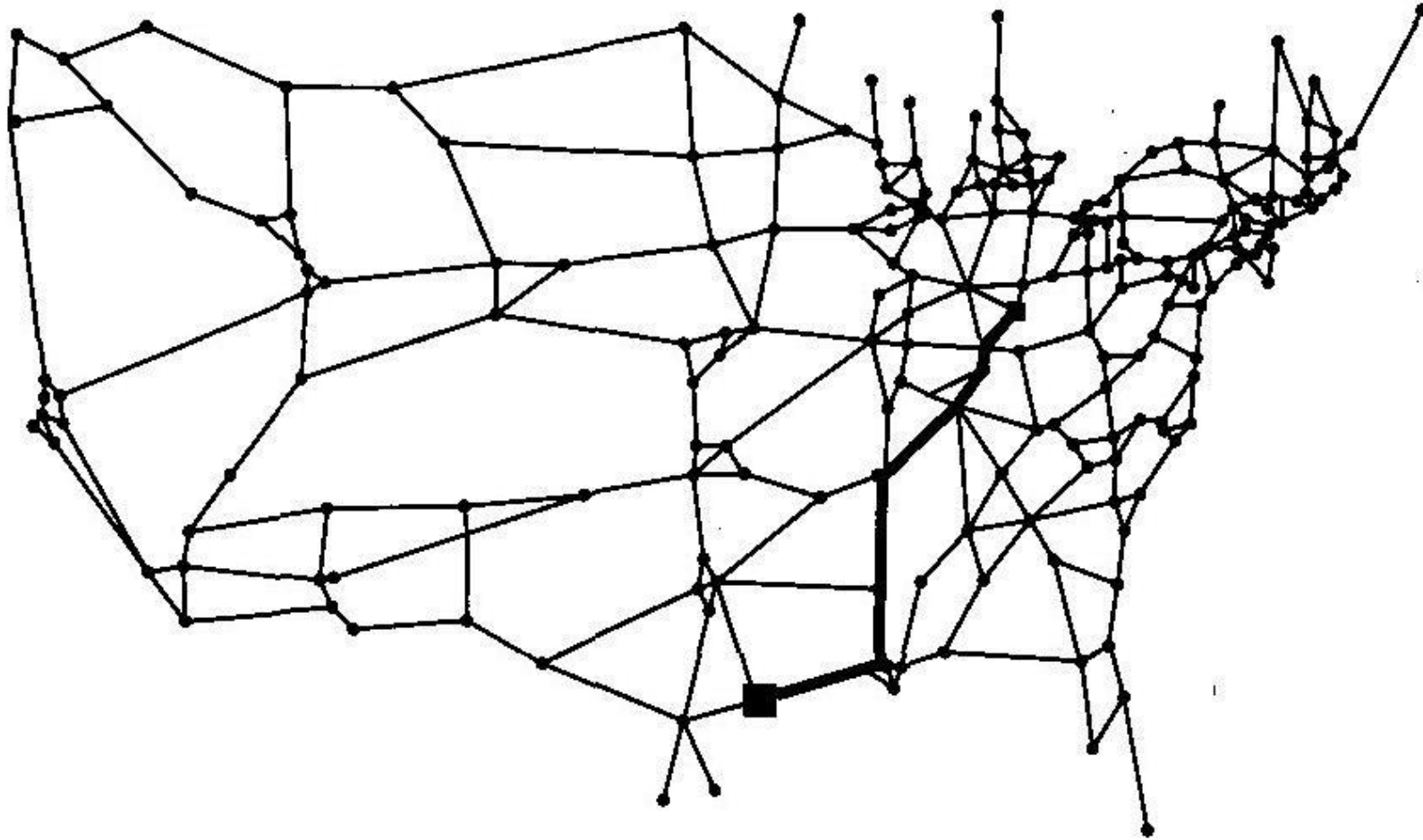


If these nodes already are tree-nodes,  
then we will not pick  $v$  from Q before  $v_{k+1}$ ,  
and when we picked  $v_k$ , we correctly set that values  $g(v_{k+1})$ , and  $\text{parent}(v_{k+1})$

and  $v_{k+1}$  is in Q

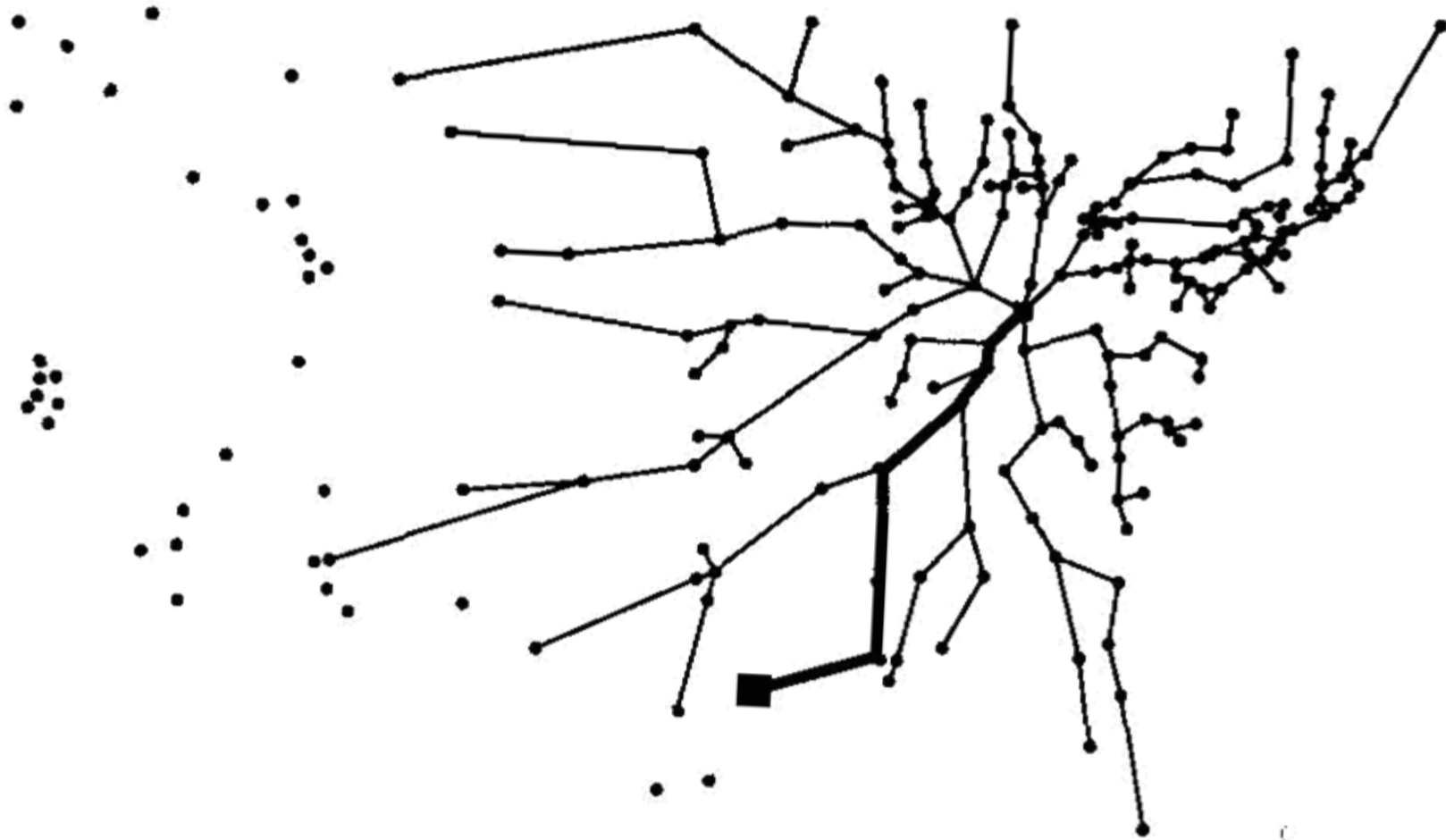
- The calculations on the previous slides show that *monotonicity implies* that  $f(v_{k+1}) \leq f(v)$
- We must in fact have  $f(v_{k+1}) = f(v)$
- If  $f(v_{k+1}) < f(v)$ , then we would not have picked  $v$  from Q before  $v_{k+1}$ , which we assumed to get a contradiction
- Therefore, we must have  $f(v_{k+1}) = f(v)$ , and  $g^*(v) = g(v)$  (the algorithm calculates the correct value for  $g(v)$ ), as shown by the monotonicity calculations on the previous slide

# A\*-search vs Dijkstra



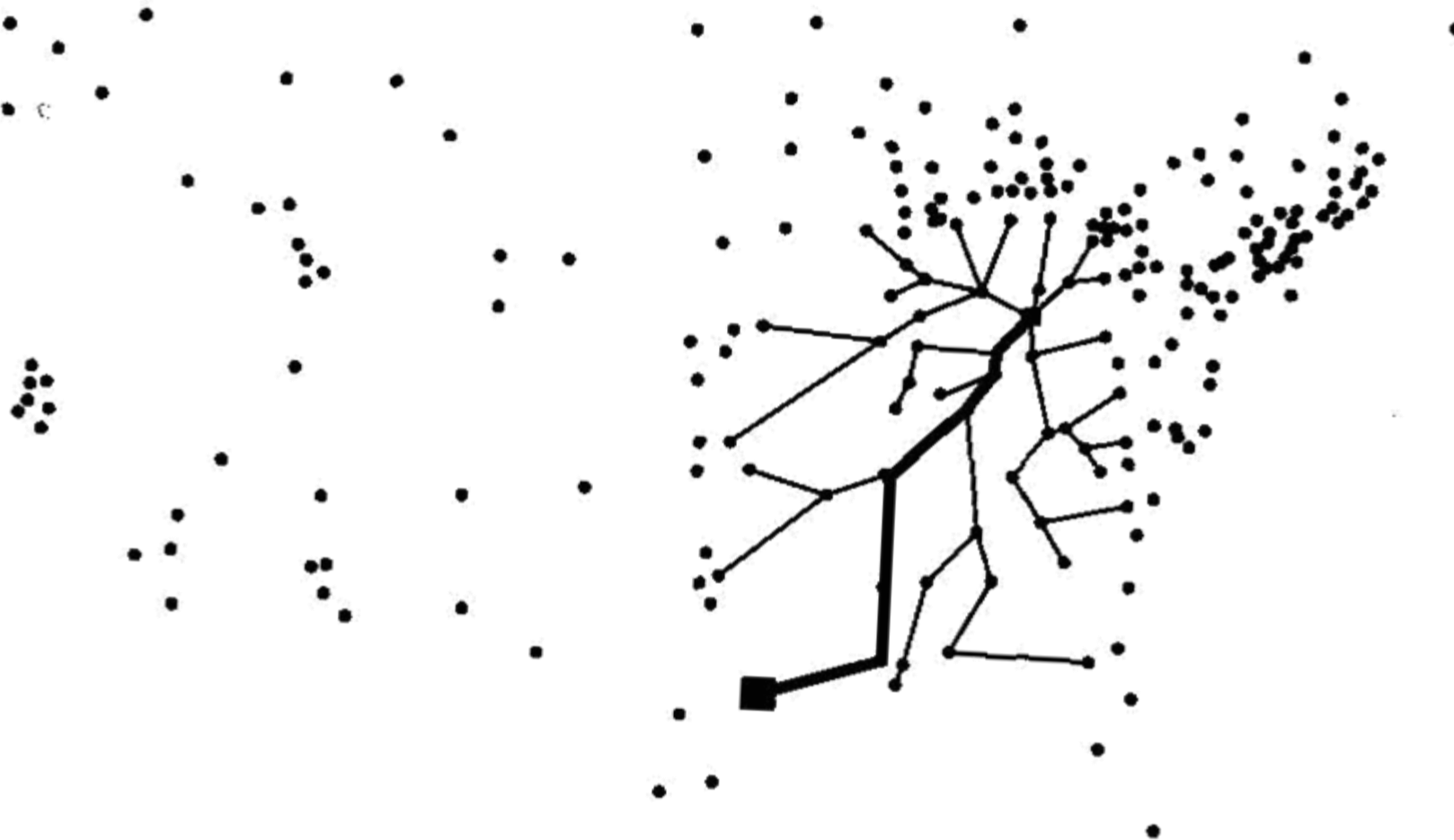
American highways. Shortest path Cincinnati - Houston is marked.

# A\*-search vs Dijkstra



The tree generated by Dijkstra's algorithm (stops in Houston)

# A\*-search vs Dijkstra



The tree generated by the A\*-algorithm with the monotone  $h(v)$ :  
 $h(v)$  = the “geographical” distance from  $v$  to the goal-node (Houston).