Search in State-spaces
13/9 – 2016

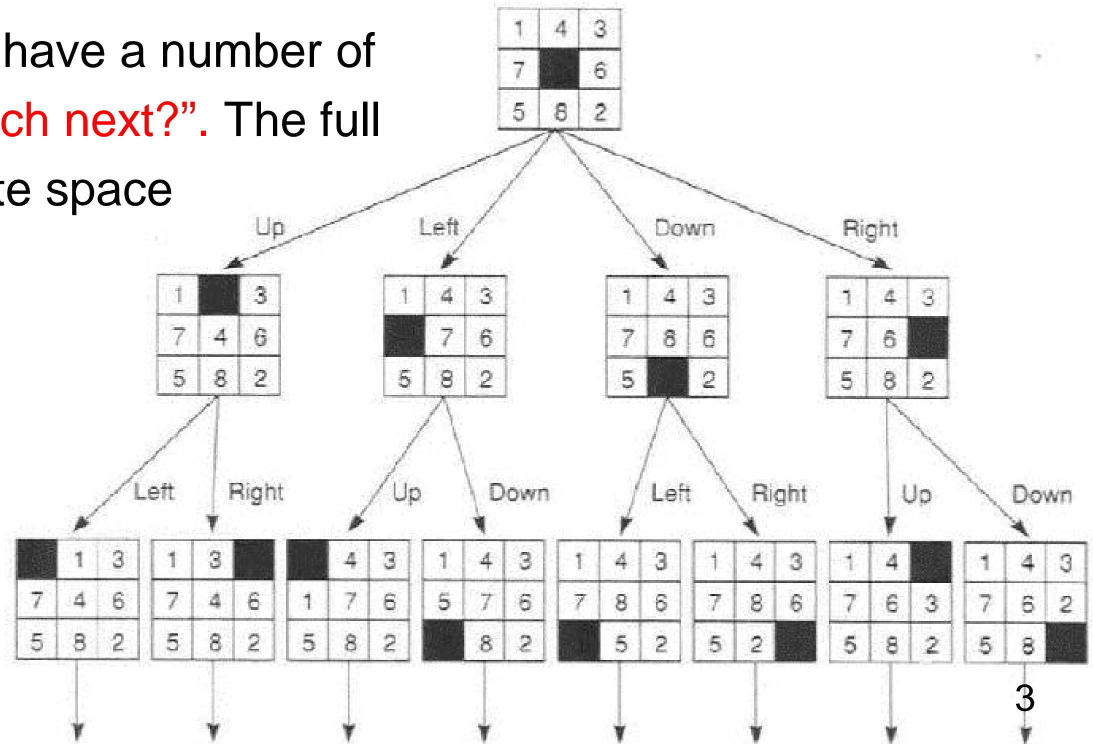
- Backtracking (Ch. 10)
- Branch-and-bound (Ch. 10)
- Iterative deepening (only on these slides, but still part of the curriculum!)
- A*-search (Ch. 23)

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
  - It 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 (heuristric), and always proceed from the currently most promising one. Natural to use a priority queue to choose next node.
- Iterative deepening
  - 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 INF2220 etc.)
- A*-search (Ch. 23)
  - Is similar to branch-and-bound, with heuristics and priority queues
  - Can also be seen as an improved version of Dijkstra's algorithm.

State-spaces and Decision Sequences

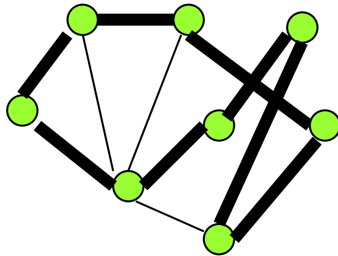
- The *state-space* of a system is the set of states in which the system can exist. Figure below: Each constellation of an 8-puzzle is a state.
- Some states are called *goal states*. That's where we want to end up. No goal state is shown below.
- Each *search algorithm* will have a way of traversing the states, and these are usually indicated by directed edges, as is seen on the figure below.
- Such an algorithm will usually have a number of decision poins: "Where to seach next?". The full tree with all choices is the state space tree for that algorithm.
- Thus, different algorithms will have different state space trees. See the following slides.
- Main problem: The state space is usually *very* large

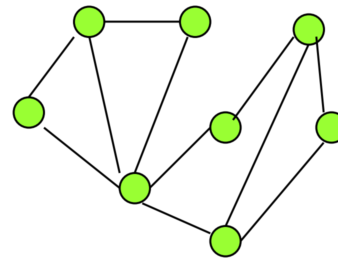


'Models' for decision sequences

- 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



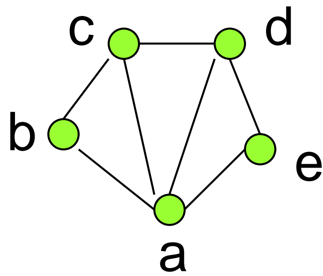


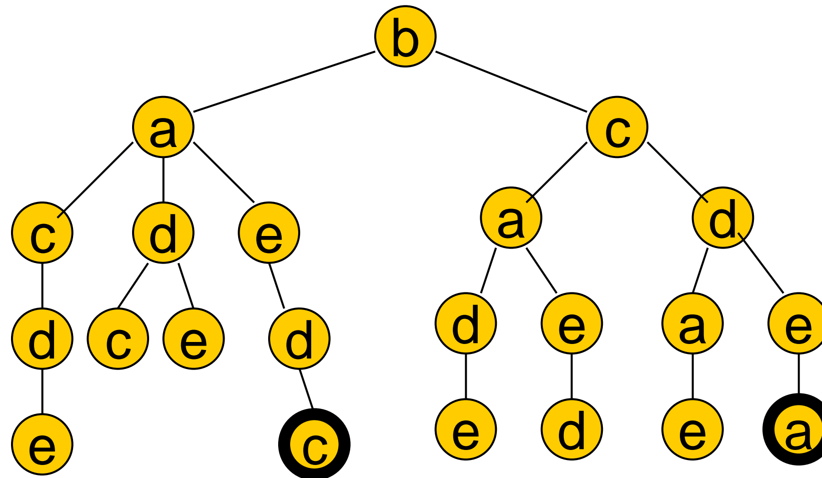
This graph obviously has no Hamiltonian Cycle

- There are (at least) two natural decision sequences:
  - Start at any node, and try to grow paths from this node in all possible ways.
  - Start with one edge, and add edges as long as they don't make a cycle with already chosen edges.
- These lead to different state space trees (see next slides).

Models for decision sequences

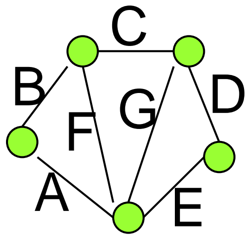
- A tree structure formed by the first decision sequence:
  - Choose a node and try paths out from from that node.
    - Possible choices in each step: Choose among all unused nodes connected to the current node by an edge.

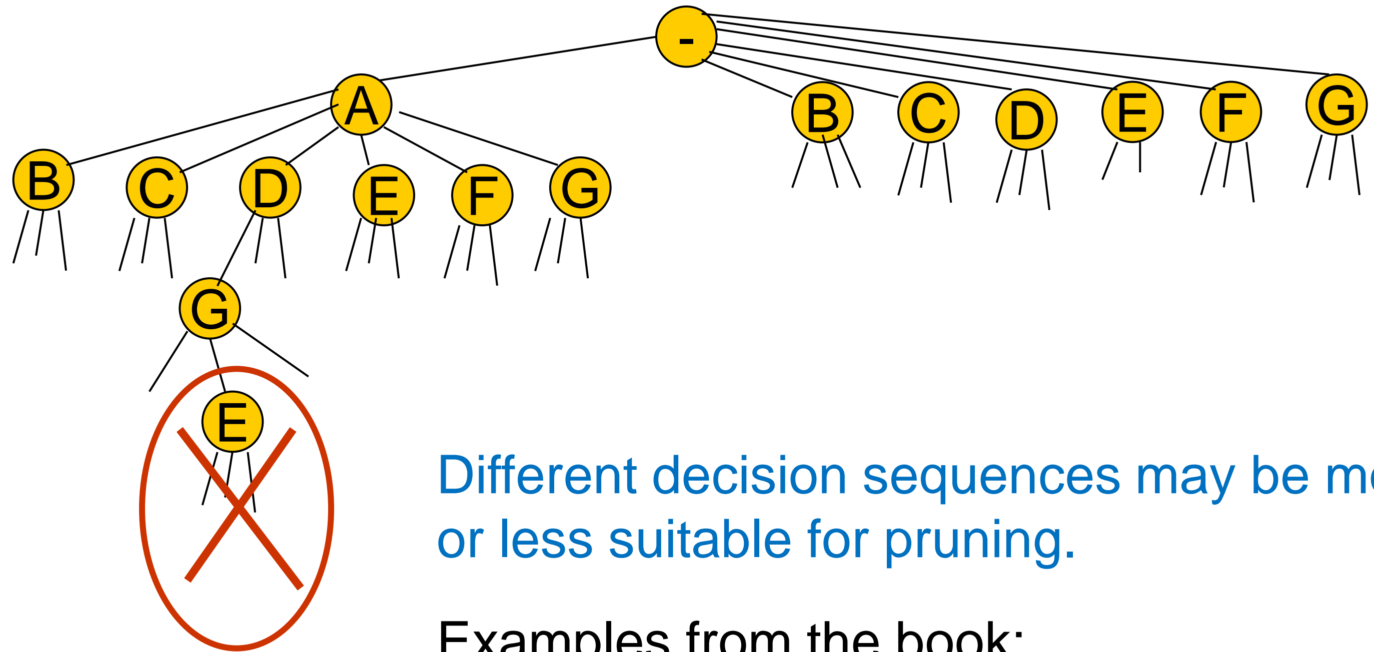




Models for decision sequences

- A tree structure formed by the second model:
  - Start with one edge, and add edges as long as they don't make a cycle with already chosen edges.

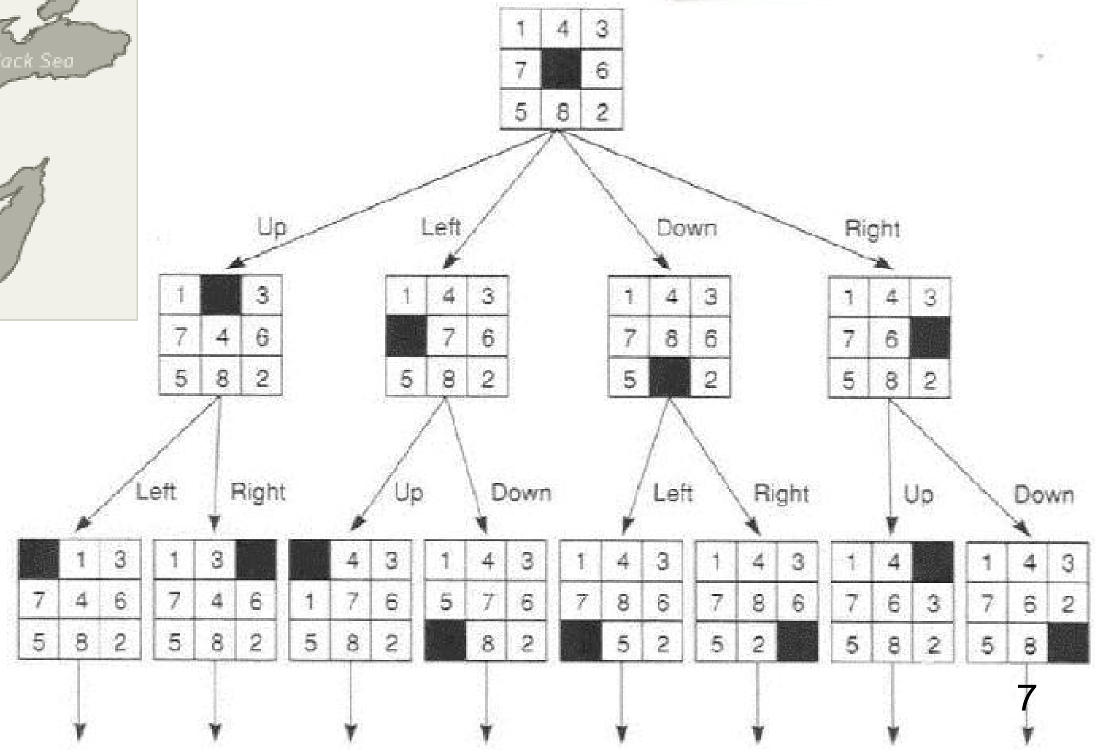




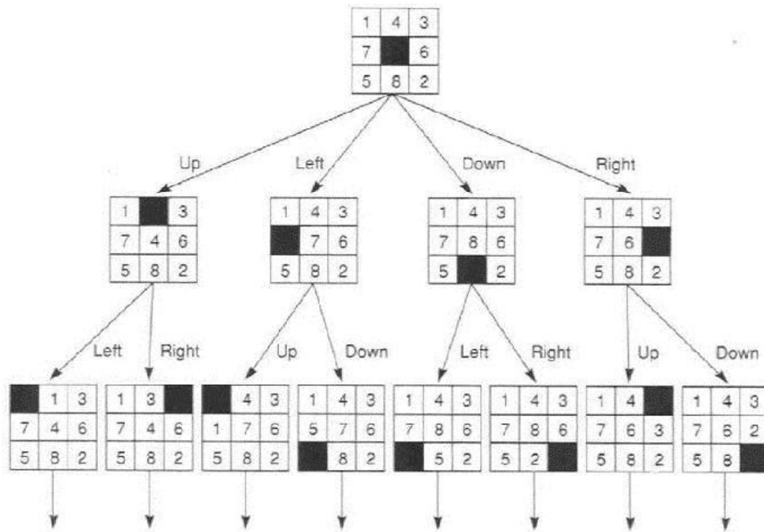
Different decision sequences may be more or less suitable for pruning.

Examples from the book:
Figure 10.3 and 10.4 (Subset sum)
Page 719 (8-puzzle, a small version of the 15-puzzle)

State spaces and decision sequences

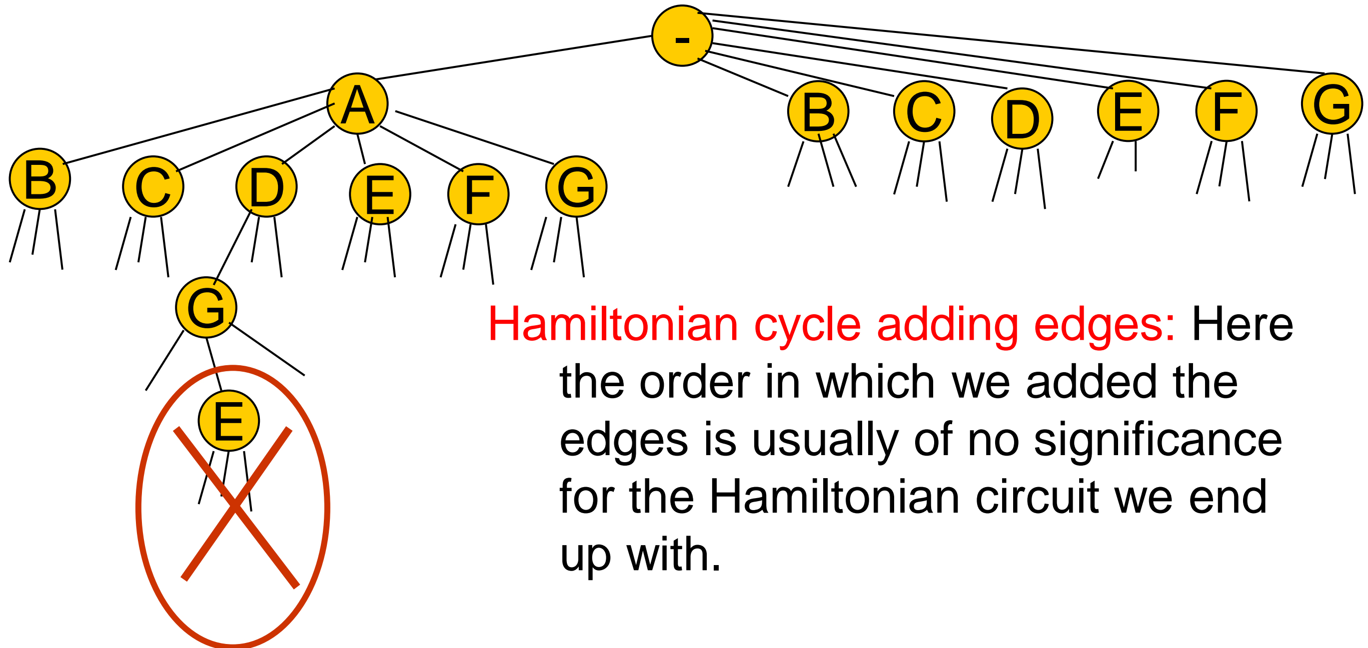


Some times the path leading to the goal node is an important part of the solution



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

But here it is not:



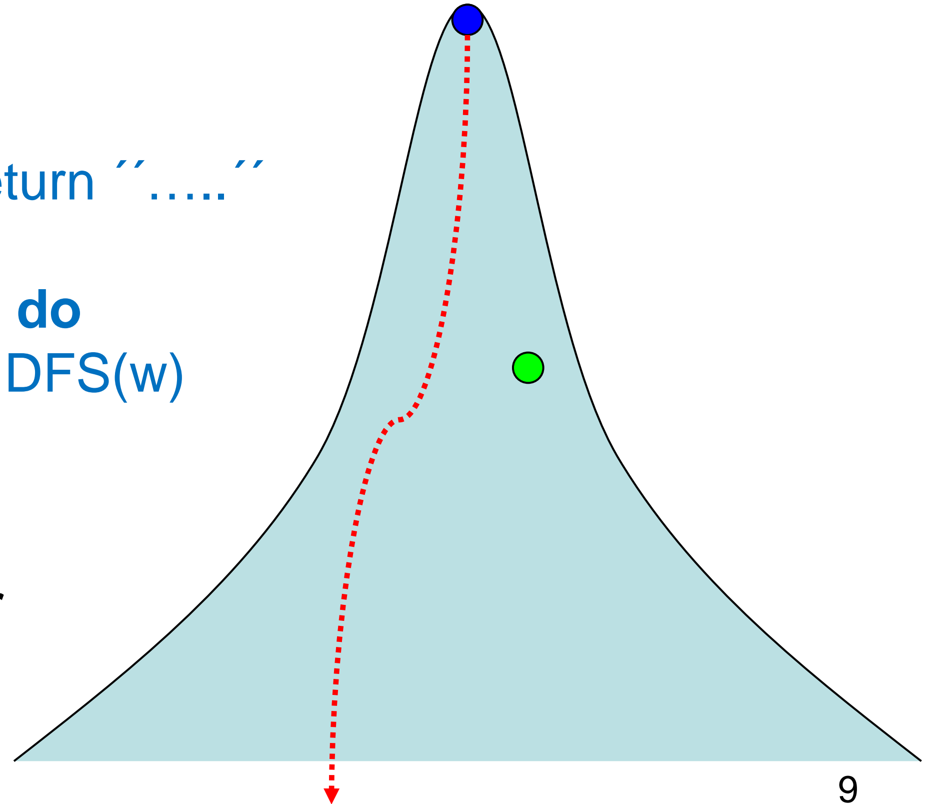
Hamiltonian cycle adding edges: Here the order in which we added the edges is usually of no significance for the Hamiltonian circuit we end up with.

Backtracking and Depth-First-Search

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

```
procdure 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)
    od
}
```

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

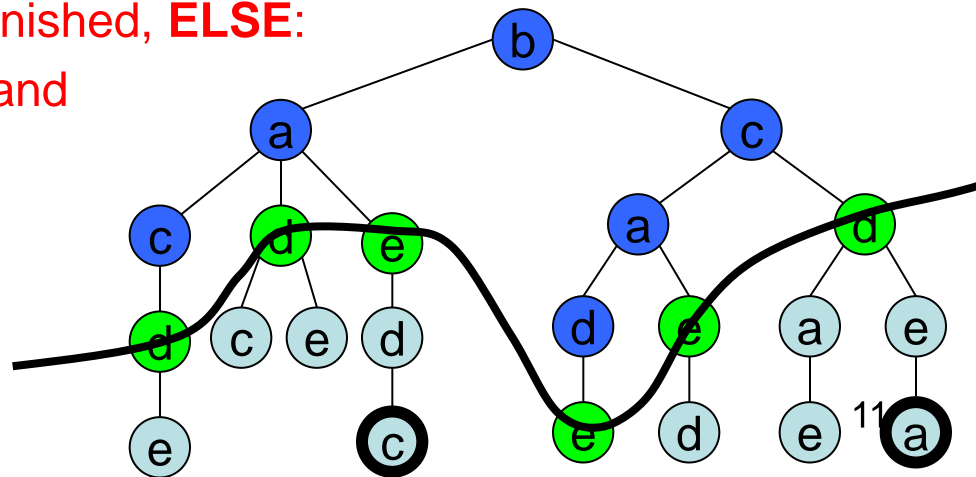


Backtracking and Depth-first-search

- 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(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. This is important, as an exhaustive search usually requires *exponential time*, and we must use all tricks we can find to limit the execution time.
- Main pruning principle: Don’t enter subtrees that cannot contain a goal node. But the difficulty is to find where this is the case.

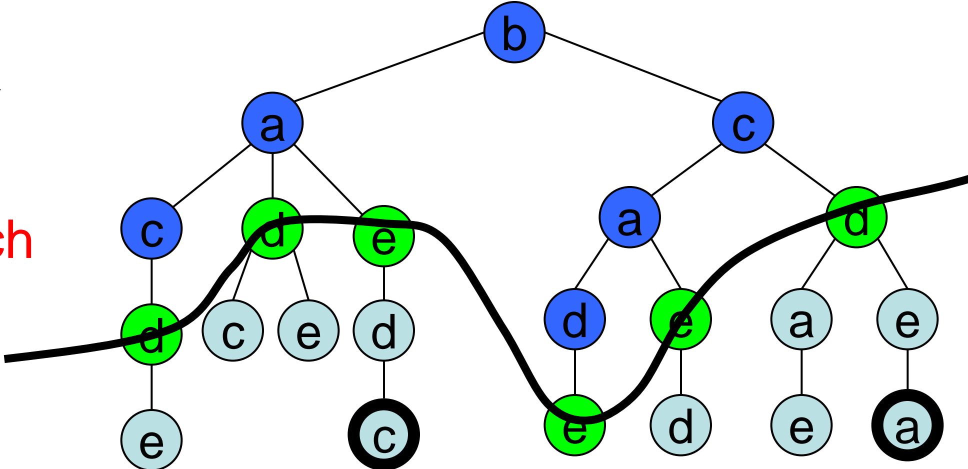
Branch-and-bound / Breadth-First-Search

- 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"* (green) 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 (green) 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.
  - If N is a goal node, then we are finished, **ELSE**:
  - Take N out of the Live-Node set and insert it into the finished nodes.
  - Insert all children of N into the Live-Node set.
  - BUT: if we are searching a graph, only insert unseen ones



Branch-and-bound / Breadth-First-Search

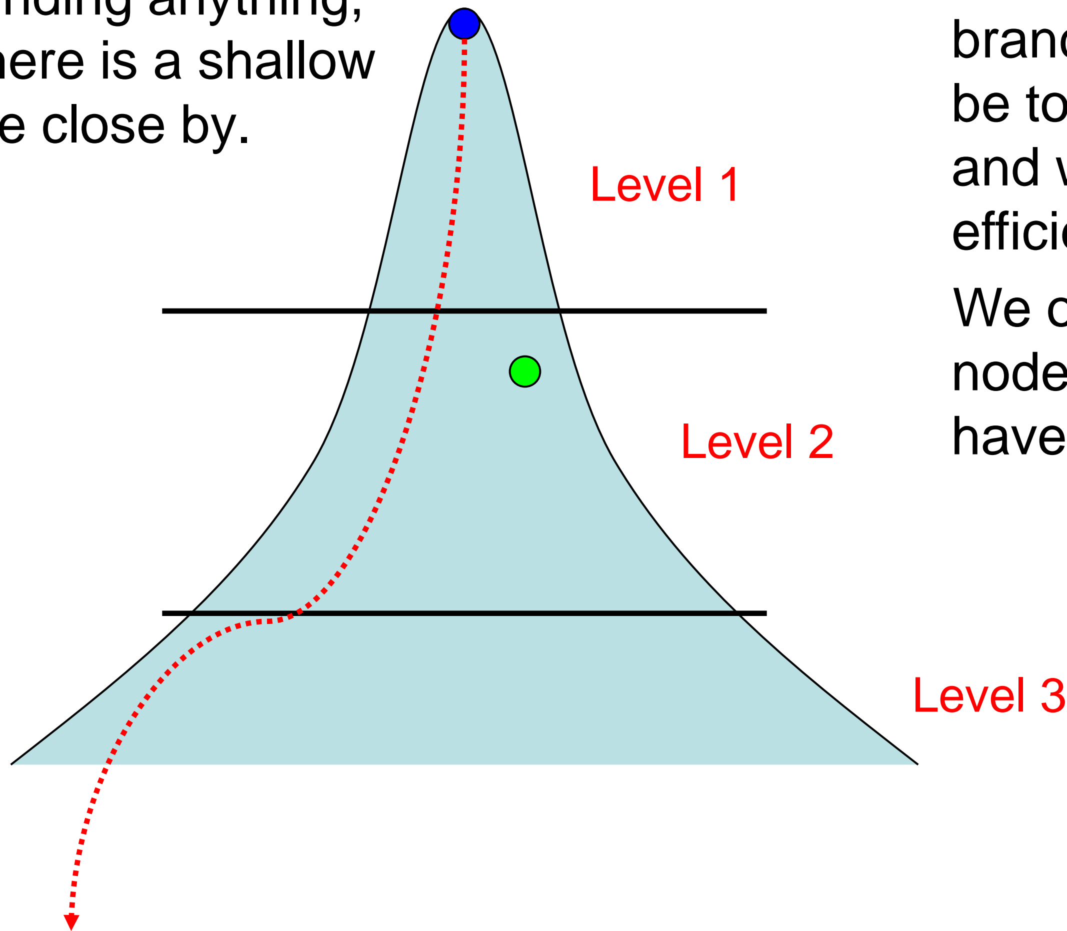
- Three strategies:
  - *The Live-Node set is a FIFO-queue*
    - *We get traditional breadth first*
  - *The Live-Node set is a LIFO-queue*
    - The order will be similar to depth-first, but not exactly
  - *The LiveNode queue is a priority queue,*
    - We can call this priority search
    - If the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



Iterative deepening

Not in the textbook, but included in curriculum!

A drawback with DFS is that you can end up going very deep in 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 facor, this will not be too much extra work, and we are always memory efficient.

We only «test for goal nodes» at the levels we have not been on before.

Iterative deepening

Assignment next week:

Adjust the DFS program to do iterative deepening:

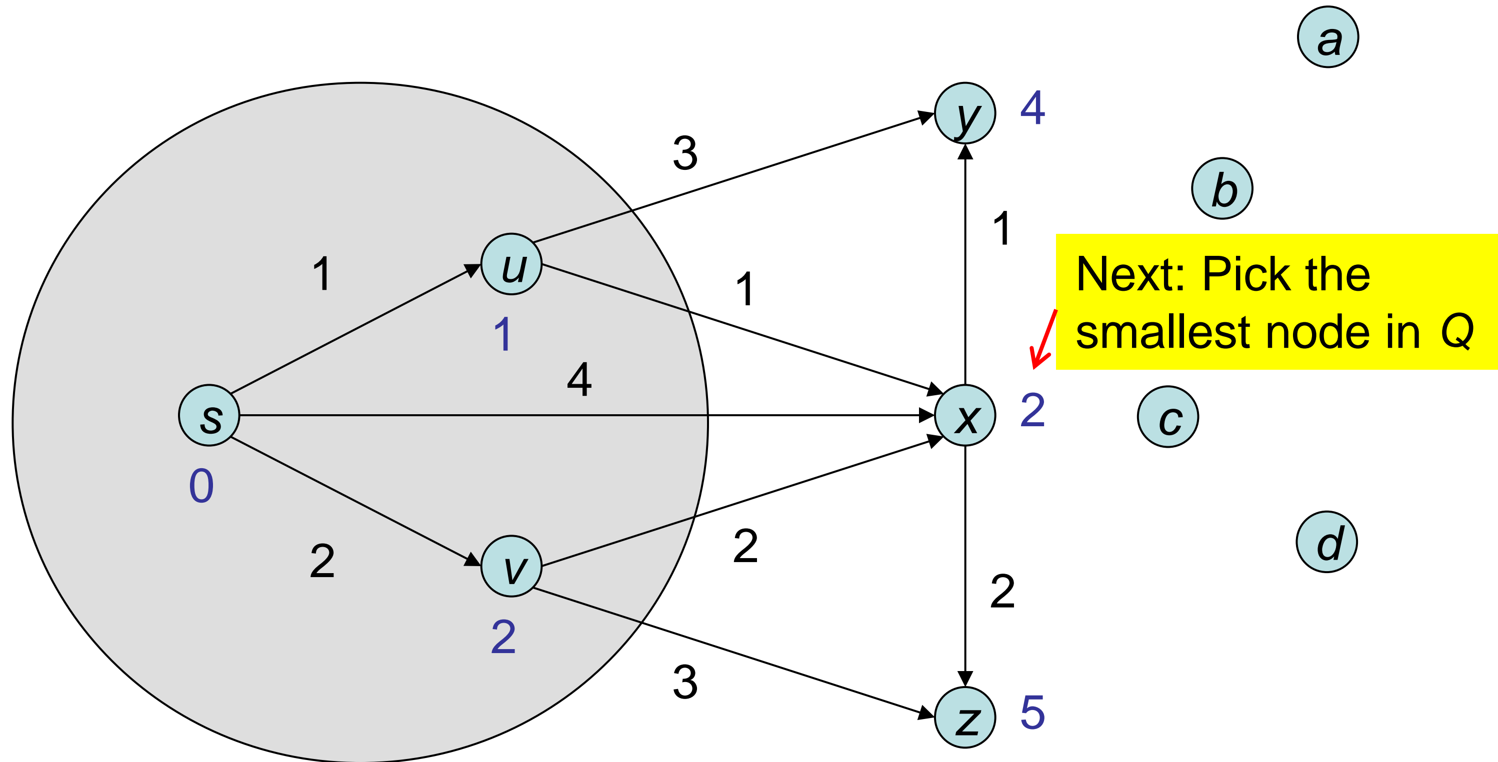
```
procedure DFS(v)
{
        If <v is a goal node> then return ´´…..´´
        v.visited=TRUE
        for <each neighbor w of v> do
                if not w.visited then DFS(w) fi
        od
}
```

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.

We move to Ch. 23, and first look at good old:

Dijkstra's algorithm for single source shortest paths in directed graphs



«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        // Initialisation
     v.dist := ∞                      //  Marks as unseen nodes
     v.previous := NIL                // Pointer to remeber the path back to source
  od
  source.dist := 0                    // Distance from source to itself
  Q := { source }                     // The initial priority queue only contains source
  while Q is not empty do
    u := extract_min(Q)               // Node in Q closest to source.  Is removed from Q
    for each neighbor v of u do       // Key in prio-queue is distance from source
      x = length(u, v) + u.dist
      if x < v.dist then              //  Nodes in the "tree" will never pass this test
        v.dist :=  x
        v.previous := u               // Shortest path "back towards the source"
      fi
    od
  od
end
```

Could already here discard nodes in the tree

A*-search (Hart, Nilsson, Raphael 1968)

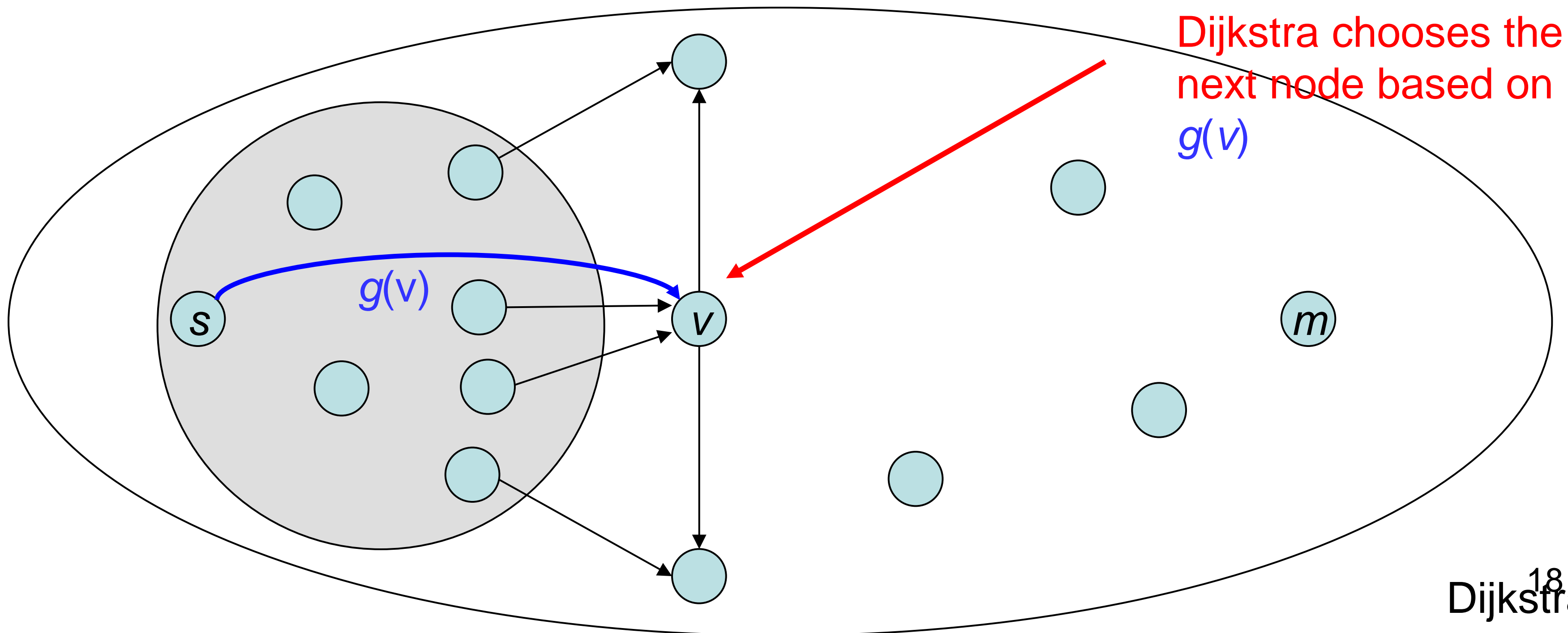
- Backtracking / depth-first, LIFO / FIFO, branch-and-bound, breadth-first and Dijkstra´s algorithm only use local information when choosing the next step.
- A*-search is simular to Dijkstra´s algorithm, but it uses a global heuristic (”qualified guess”) to make better choices from Q in each step.
- Widely used in AI and knowlegde 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 (h for *heuristic*).
  - The value we use for choosing the best next node is now:

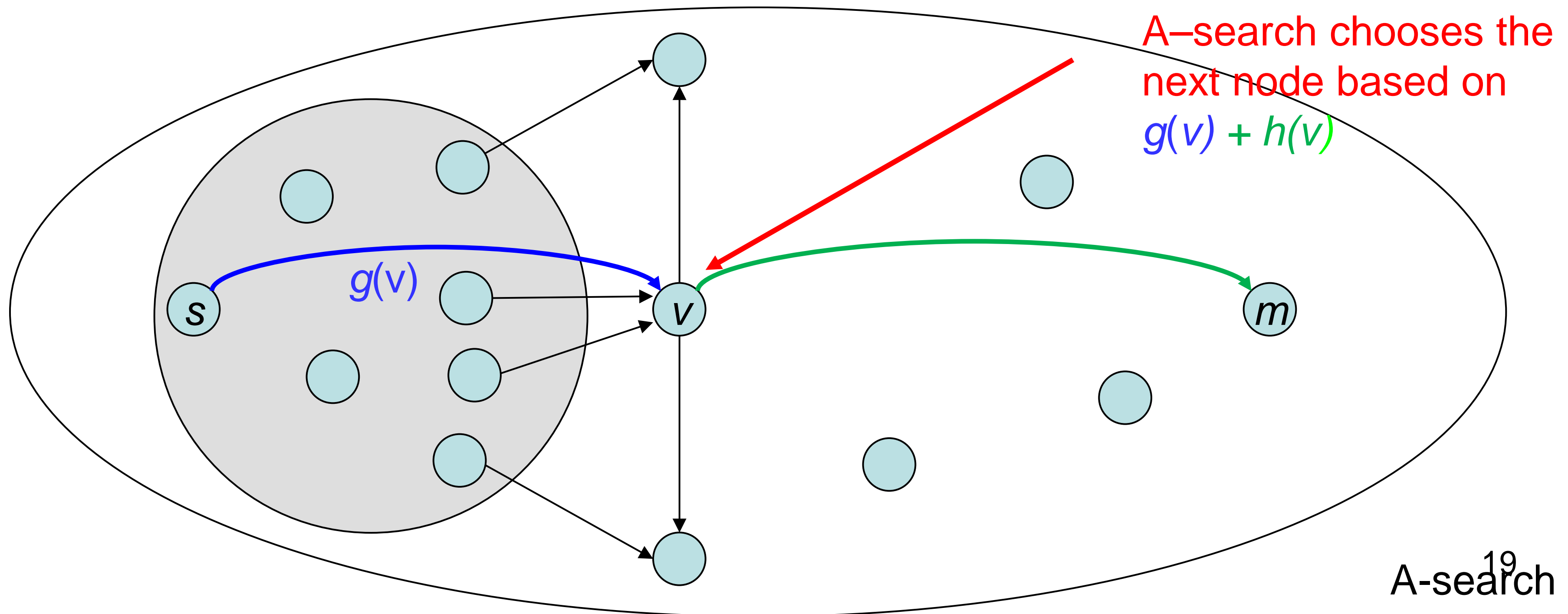
    $f(v) = g(v) + h(v)$ (while Dijkstra uses only $g(v)$)
  - Thus, we get a priority-first search with this value as priority.



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 (h for *heuristic*).
  - The value we use for choosing the best next node is now:

    $f(v) = g(v) + h(v)$ (while Dijkstra uses only $g(v)$)
  - Thus, we get a priority-first search with this value as priority.



A*-search and heuristics

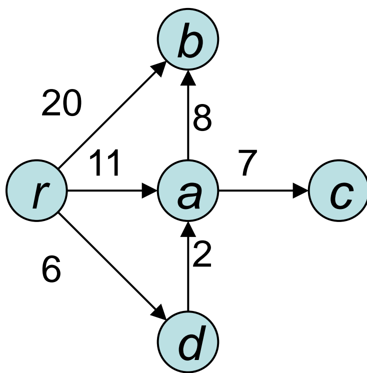
- If we know that $h(v)$ is never larger than the actual shortest path to the closest goal node
  - and we use the A-search algorithm described on the previous slides, we will eventually get the correct result (this is not proven here)
- However, we will often have to go back to nodes that we «thought we were finished with» (nodes in the tree will have to go back into the priority queue, as we may later find an even shorter path to them)
  - And this usually results a lot of extra work
- Dijkstra´s algorithm never moves a tree-node back into the queue, and still get the correct answer. We will put requirements on $h(v)$ so that this will also be true for the A-search algorithm above.

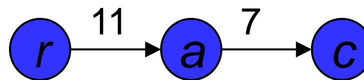
Exercise
(”admissible heuristic”)

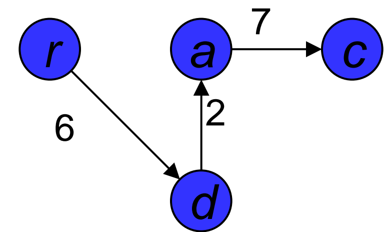
- Study the example in figure 23.5 (the textbook, page 723). It demonstrates some of what is said on the previous slide:

  If you do not move nodes back into Q from the «tree» (when a shorter path is found), you may not find the shortest path

- The drawings below is from that example:







v	r	a	b	c	d
$h(v)$	-	23	20	15	29

A*-search and monotone heuristics

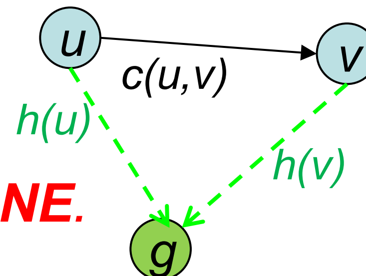
The function $h(v)$ should be an estimate of the distance from v to the closest goal node. If $h(v)$ is never larger than the shortest distance to a goal, we know that the search will terminate, but maybe slowly.

However, we can restrict $h(v)$ further, and get more efficient algorithms.

- the function $h(v)$ must then have the follwing properties:
  - (As before) The function $h(v)$ is never larger that the real distance to the closest goal-node.
  - $h(g) = 0$ for all goal nodes g.
  - And a sort of *"triangule 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) The criteria for *monotonicity* on *h:*
  - If there is an edge from v to w with weight $c(v,w)$, then we must always have $h(v) \leq c(v,w) + h(w)$
  - Every goal node g must have $h(g) = 0$.

.

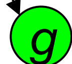
- A nice thing here is that if these two requirements are fulfilled:
  - then the first requirement (that $h(v)$ is never larger than the real shortest distance to a goal) is automatically fulfilled.

- Sketch of a proof: We assume that $u \rightarrow v \rightarrow w \rightarrow m$ is a shortest path from u to a goal state g. We set up the inequalities we know:

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

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

$$h(w) \leq c(w,g) + h(g) = c(w,g)$$



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

Relation to Dijkstra's algorithm

- If we use $h(v) = 0$ for all nodes, this will be Dijkstra's algorithm
- By using a better heuristic we hope to work *less* with paths that do not lead to solutions (a shortest path), so that the algorithm will run faster.
- However, with a heuristic we will remove nodes from Q in a differnt order than with Dijkstra´s algorithm.
- Thus, we can no longer know 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.

See slides below.

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

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

A*-search – the data for the algorithm

Will be studied as an exercise

- We have a directed graph G with edge weights *c(u,v),* a start node *s*, a number of goal-nodes, and finally a monotone heuristic function *h*(*u*), (often set in all nodes during initialisation, 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 intto 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

Will be studied as an exercise next week

- Q is initialized with the start node *s*, with $g(s) = 0$. (all other nodes are unseen)

- The main step, that are repeted until Q is empty:

  Find the node v in Q with the smallest f-value. We then have two alternatives:

  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 neighbours 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 $parent(w) = v$.
       – Put w into PQ .
     - 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») do *not* look at neighbours of v that are tree nodes. That this will work needs a proof, see next slides

Proof that strong A*-search works

You will not be asked about details in this proof at the exam, but you should know how the induction goes

Proof of proposition 23.3.2: We must use induction (not done in the book).

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

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 cahanged)

Def: Let generally $g^*(u)$ be the length of the 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*

$s = v_0, v_1, v_2, \dots, v_j = v$

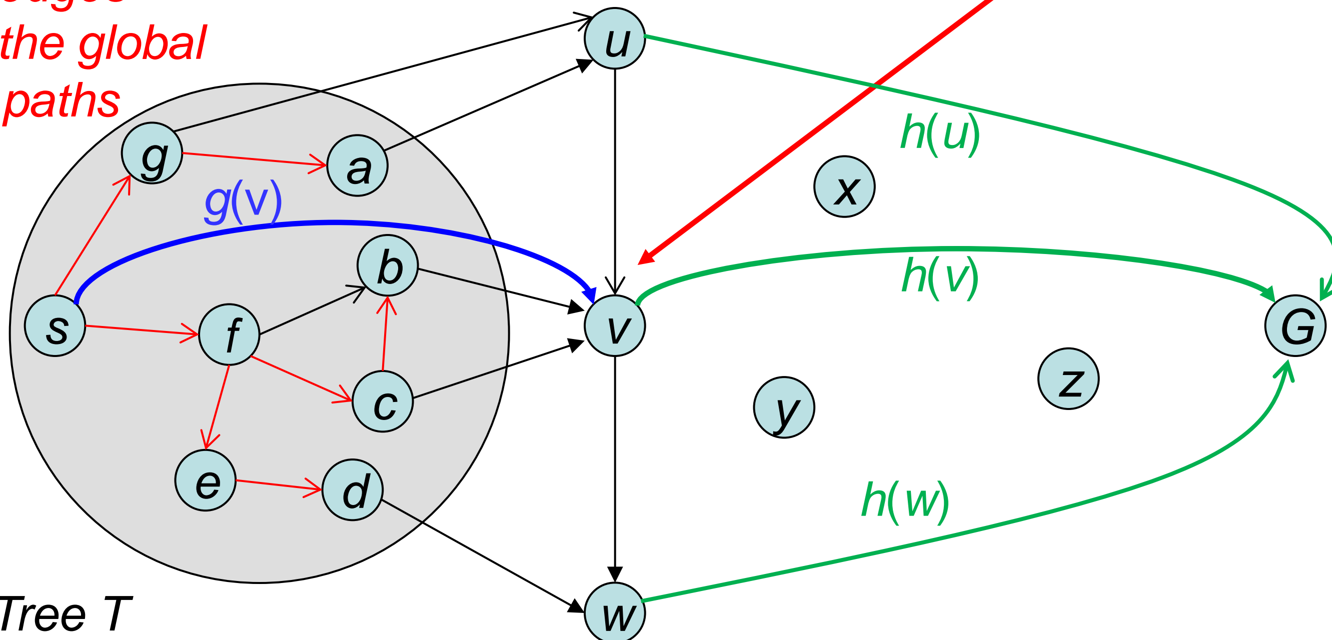
We now assume that $v_0, v_1, \dots, v_{k,}$, but not v_{k+1}, where k+1 < j, have become tree nodes when *v* is removed from *Q*, so that v_{k+1} is in *Q* when *v* is removed from *Q*. We shall show that this cannot be the case.

Illustrating the proof for A*-search

– The value we use for choosing the best next node is now:

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

- NB: We assume that the heuristic function *h* is monotone
- The important point is to show that there cannot be a shorter path to *v* that pass through another node outside the tree, e.g. through *u*.



End of proof that A*-search works

From the monotonicity we know that (for $i = 0, 1, \dots, j\text{-}1$)

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

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

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

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, …, j-1 respectively, we get

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

From the induction hypotheses we know that $g(v_k) = g^*(v_k)$, (by looking at the actions done when v_k was taken out of Q) $g(v_{k+1}) = g^*(v_{k+1})$, even if it occurs in Q.

We thus know:

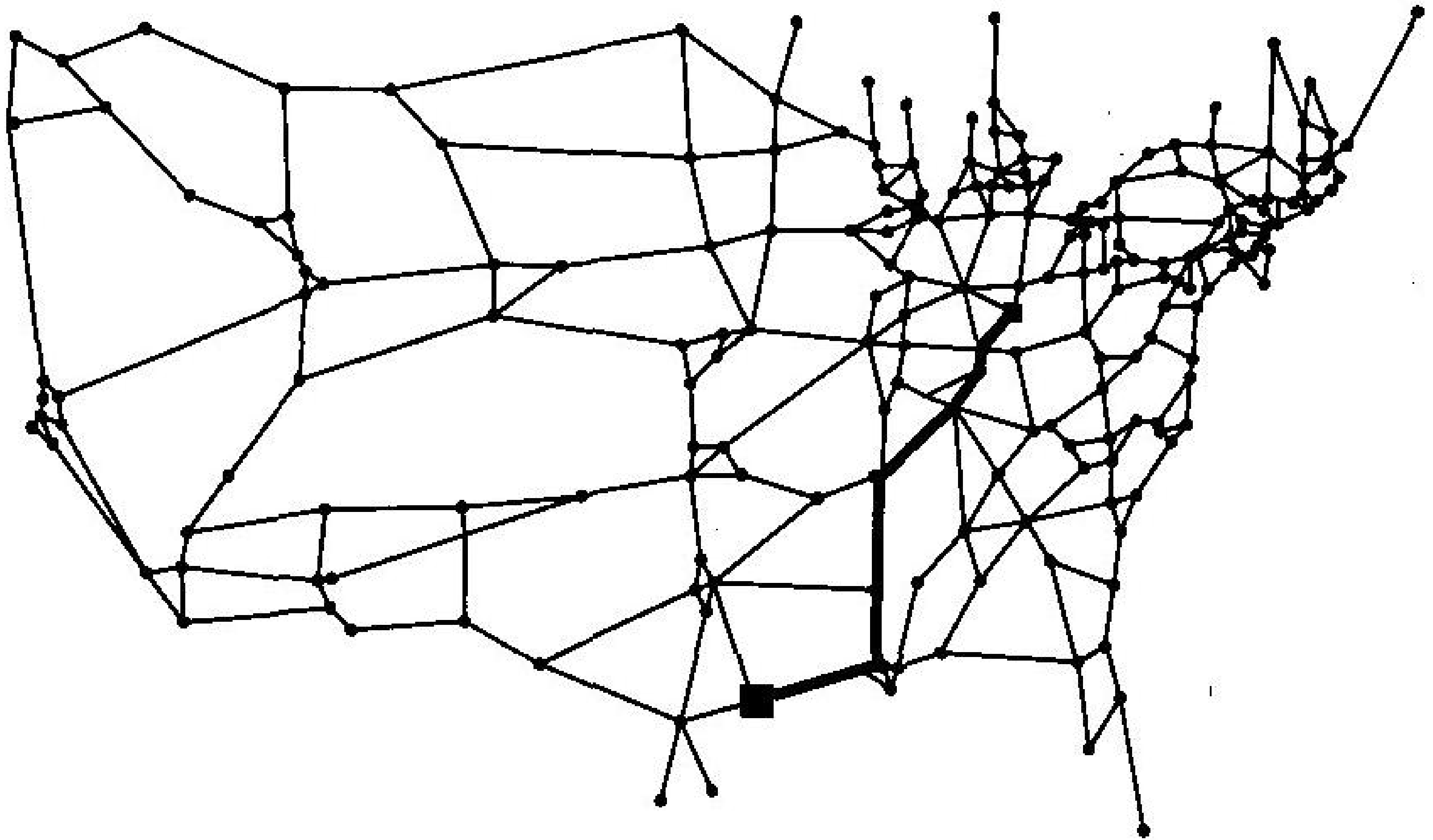
$$f(v_{k+1}) = g(v_{k+1}) + h(v_{k+1}) = g^*(v_{k+1}) + h(v_{k+1}) \leq g^*(v) + h(v) \leq g(v) + h(v) = f(v)$$

Here, all '≤' 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

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

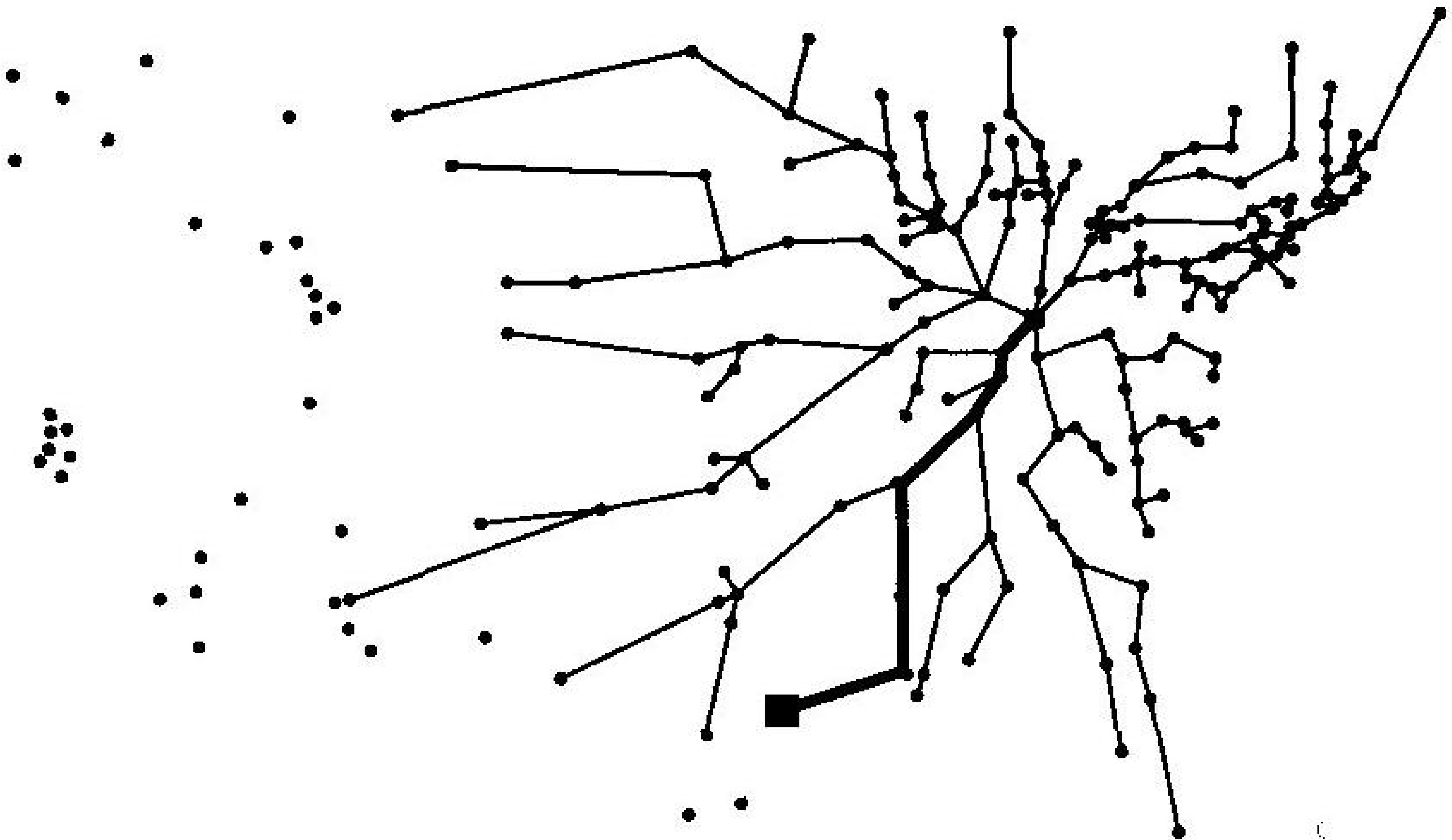
End of proof

A*-search



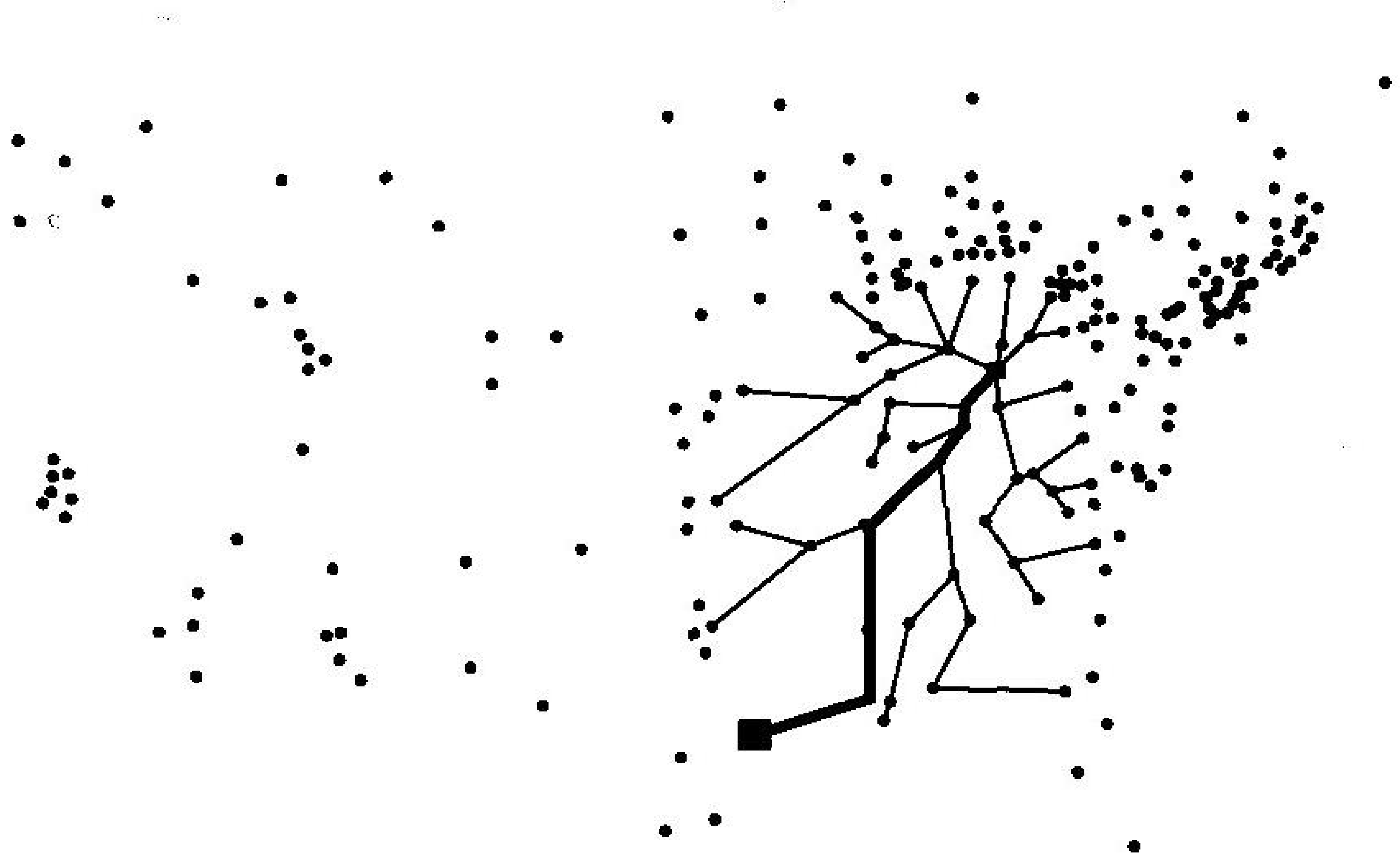
American highways. Shortest path Cincinatti - Houston is marked.

A*-search



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

A*-search



The tree generated by the A*-algorithm with the monotone *h(v):*

h(v) = the «geographical» distance from *v* to the goal-node (Houston).