Search in State-Spaces
17/9 - 2013

NB: Oblig 1 (mandatory assignment 1) is availible at the INF4130 homepage

Deadline is Friday, September 27

The topics of the day:

- *Backtracking* (Ch. 10)
- *Branch-and-bound* (Ch. 10)
- Iterative deepening (only on these slides)
- A*-search (Ch. 23)

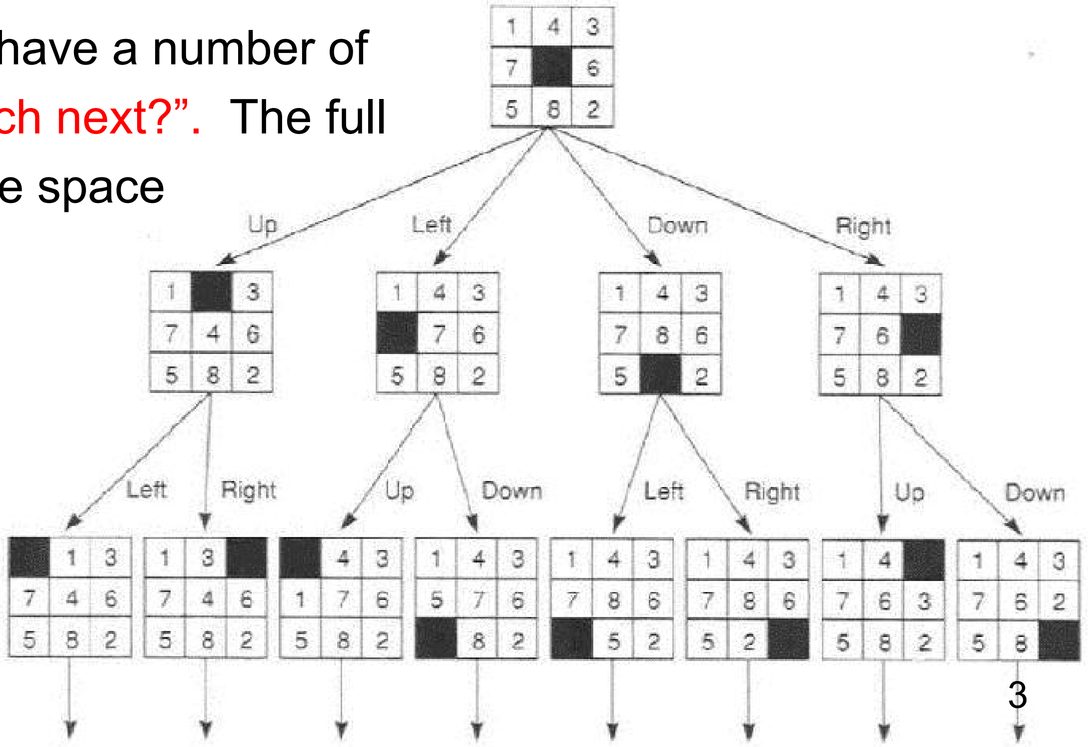
Search in State-Spaces

Ch. 10 is mainly old stuff from traditional courses
on *Algorithms and Data Structures* (e.g. INF2220 at UiO)
Ch. 23 is probably new stuff for most of you

- *Backtracking* (Ch. 10)
  - Depth-First-Search in a state-space: DFS
  - Do not need much memory
- *Branch-and-bound* (Ch. 10)
  - Breadth-First-Search: BFS
  - It needs much 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 net 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)
- 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

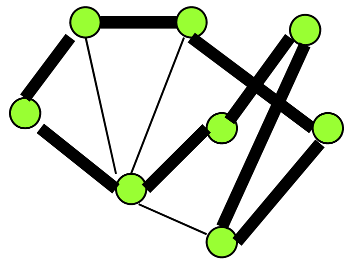
- The *state-space* of a system is the set of states in which the system can exist. Figure below: Each constellation of an 8-puzzel is a state.
- Some states are called *goal states*. That's where we want to end up. No goal state is shown below.
- Each *search algorithm* will have a way of traversing 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

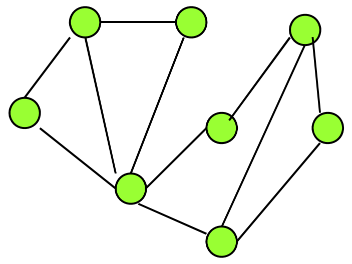


”Models” for decision sequences

- There are usually more than one decision sequence for a given problem, each leading to different state space trees.
- Example: Find, if possible, a Hamiltonian Cycle (see figures below)

Hamiltonian Cycle



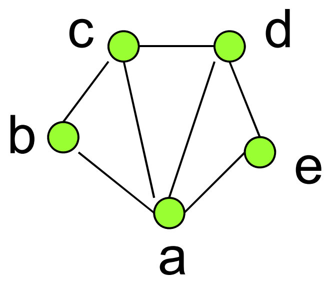


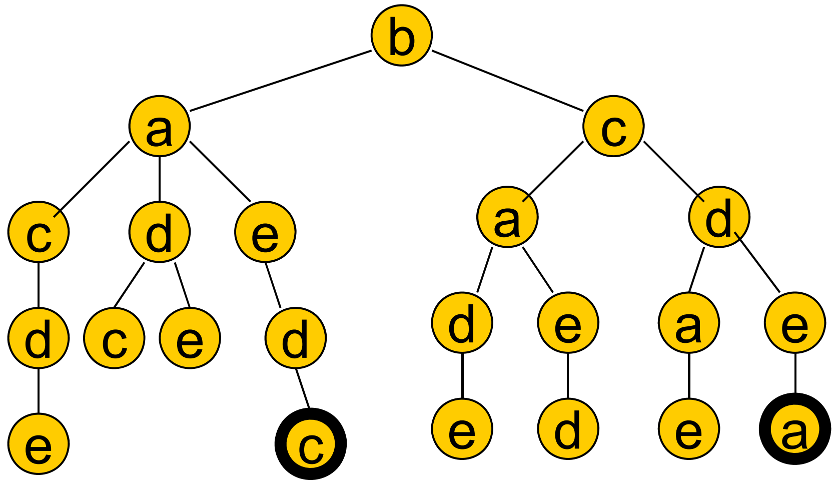
This graph obviously has no Hamiltonian Cycle

- 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 leads to different state space trees (see next slides).

Models for decision sequences

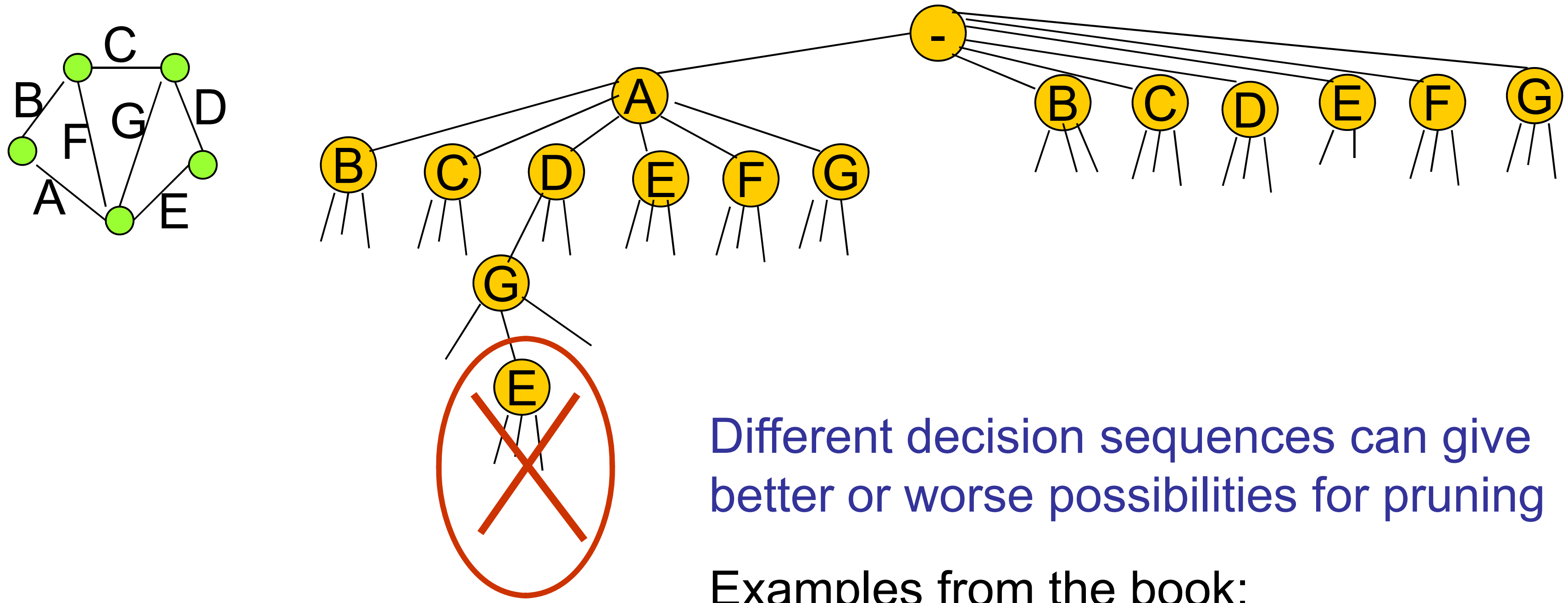
- A tree structure formed by the first decision sequence:
  - Choose a node and try pathes from this in all ways
    - Possible choices in each step: Choose among all unused nodes connected to the current node by an edge.





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.

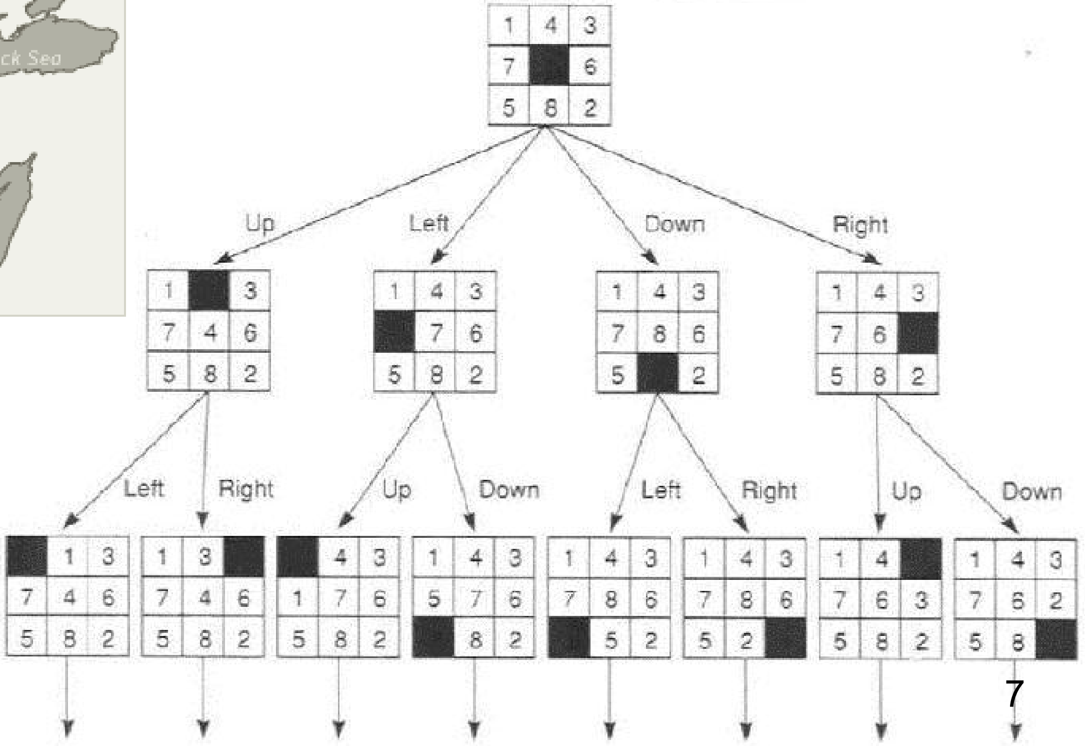


Different decision sequences can give better or worse possibilities for pruning

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

Norwegian Sea
Atlantic Ocean
Black Sea
Mediterranean Sea
Trondheim
Bergen
Oslo
Stavanger
Aberdeen
Gothenburg
Copenhagen
Helsinki
St. Petersburg
Tallinn
Stockholm
Moscow
Dublin
Manchester
Hamburg
Gdansk
Vilnius
Birmingham
Amsterdam
Hanover
Poznan
London
Dusseldorf
Berlin
Warsaw
Brussels
Frankfurt
Prague
Stuttgart
Vienna
Paris
Munich
Budapest
Zürich
Geneva
Milan
Nice
Madrid
Barcelona
Rome
Palma de Mallorca
Alicante

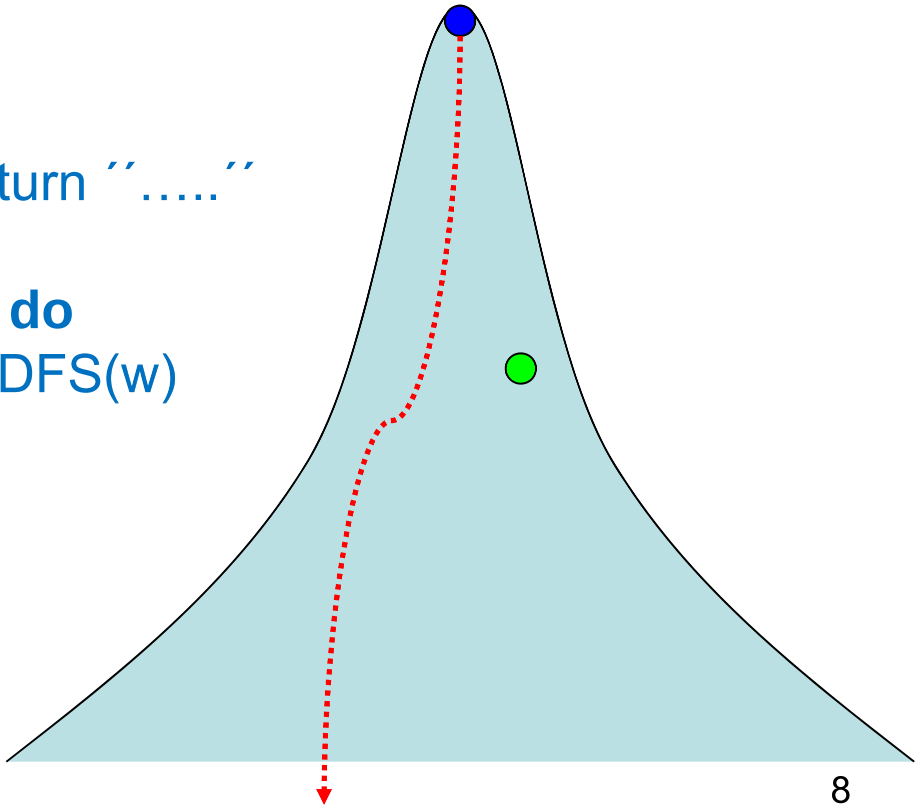


Backtracking and Depth-First-Search

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

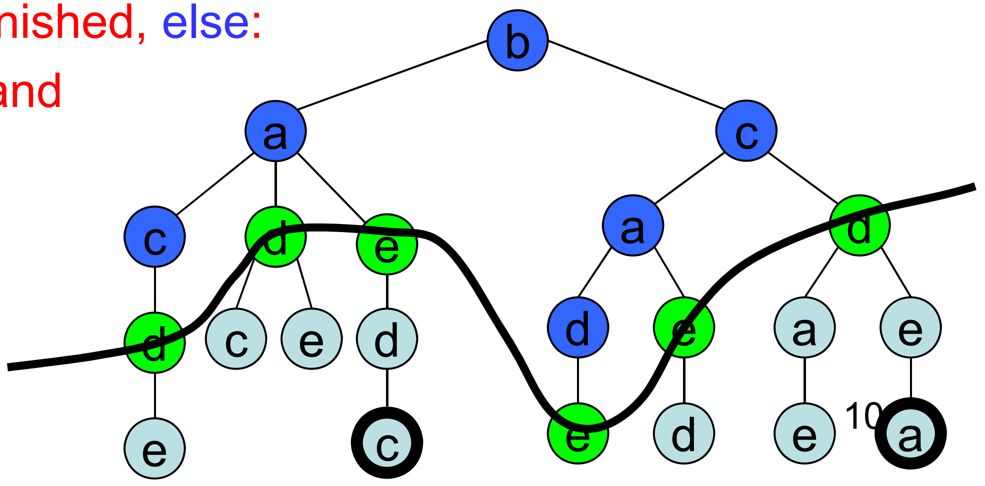


Backtracking and Depth-first-search (DFS)

- 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
- Needs little memory (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 a search usually is *time-exponentia*l, 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 true.

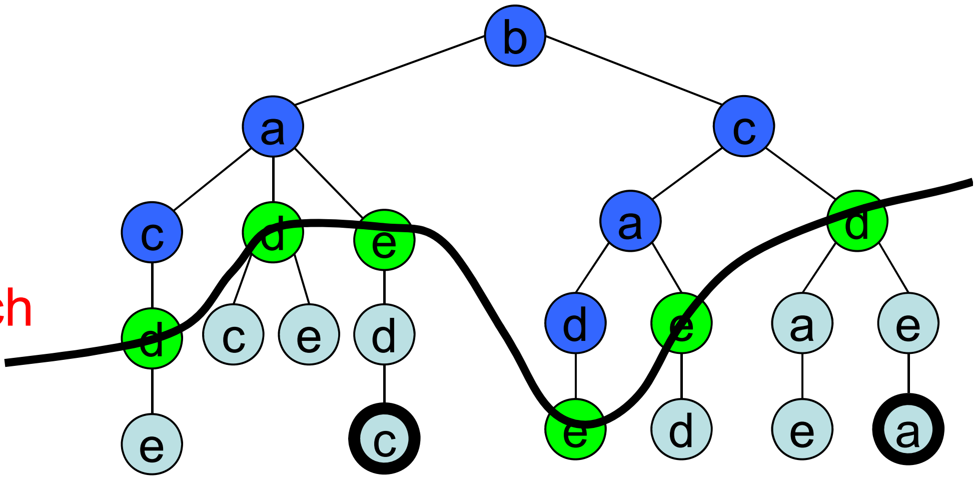
Branch-and-bound / Breadth-First-Search

- Uses some form of breadth-first-search.
- We have three sets of nodes (only the second must always be stored):
  1. The *"finished nodes"* (deep blue). Often need not be stored
  2. the *"live nodes"* (green) (seen but not explored further. Large set, must be stored).
  3. The *"unseen nodes"* (Light blue) Do we 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

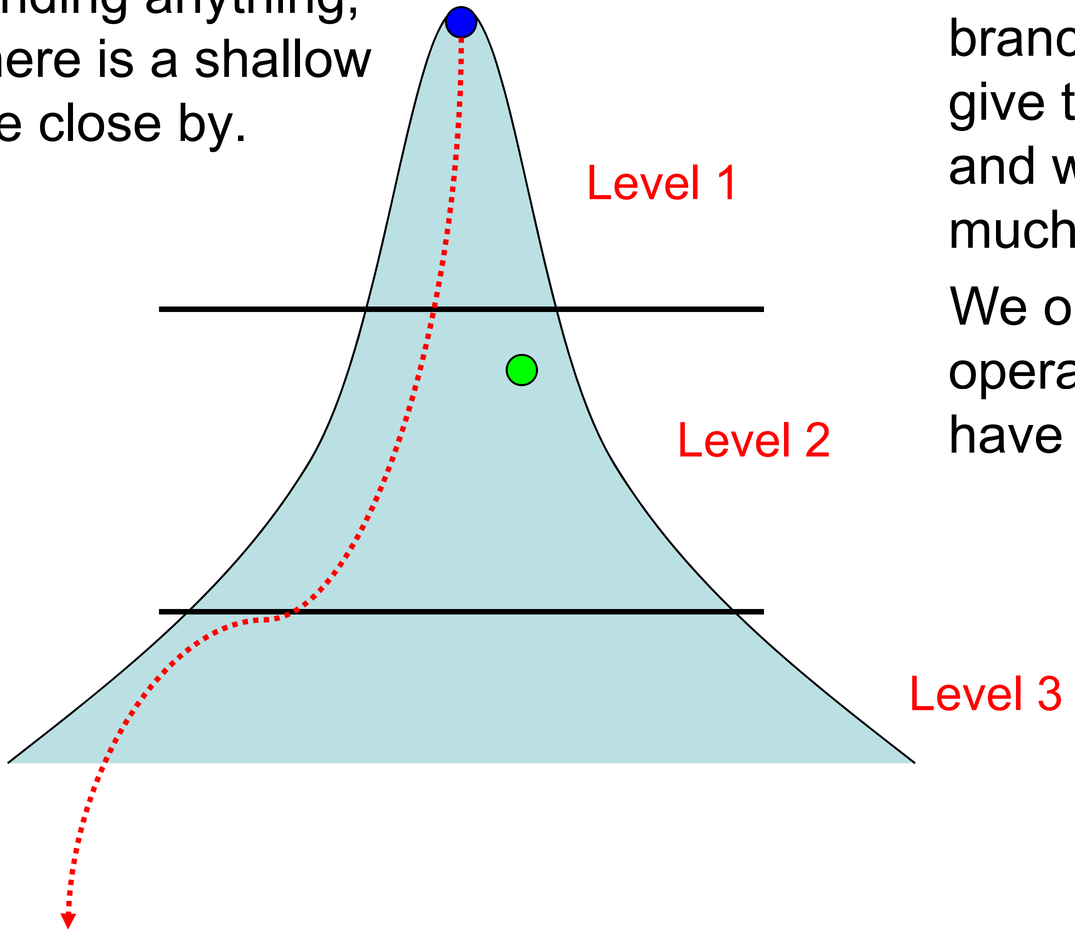
- Three strategies:
  - *The Live-Node set is a FIFO-queue*
    - *We get traditional breadth first*
  - *The Live-Node set is a LIFO-queue*
    - The order will be similar to depth-first, but not exactly
  - *The LiveNode queue is a priority queue,*
    - We can call this priority search
    - If the 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 give too much extra work, and we never have to use much space.

We only do ”real operations” at the levels we have on been before

Iterative deepening

Assignment next week:

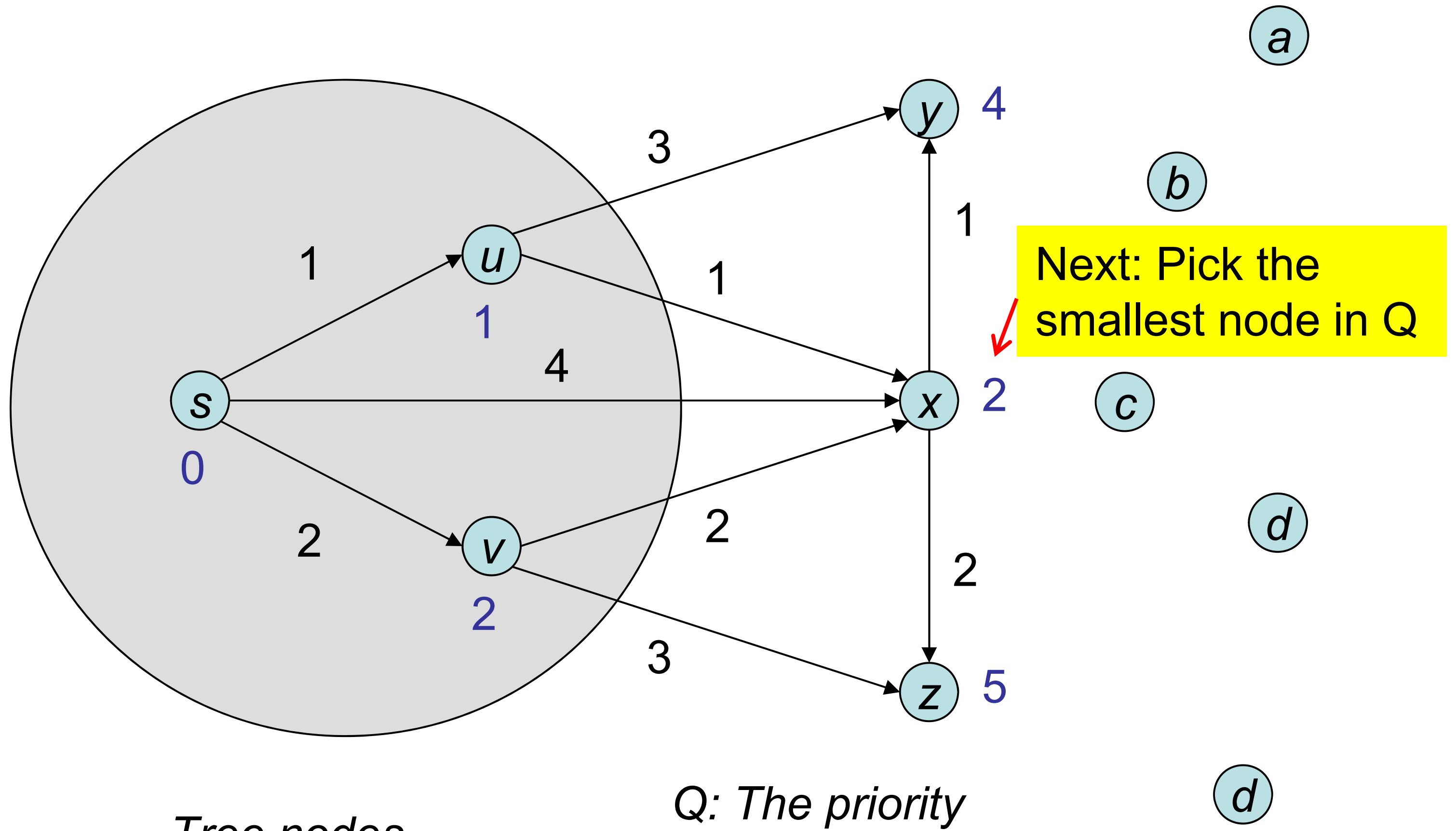
Adjust the DFS program to do iterative deepening:

```
proc DFS(v)
{
        If <v is a goal node> then return ´´…..´´
        v.visited=TRUE
        for <hver nabo w av v> do
                if not w.visited then DFS(w)
        od
}
```

NB: We should only test for goal-nodes at the ”new levels”, as we assume that this is an expensive test.

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

Dijkstra's algorithm 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 vertex v in Graph 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 , enter Q
   Q := { source }                  // S is the set of unvisited nodes
while Q is not empty do
   u := extract_min(Q)              // Node now closest to source.  Is removed from S
   for each neighbor v of u do         // Key in prio-queue is distance
      v.alt = u.dist + length(u, v)
      if v.alt < v.dist then           //  Nodes in the ”tree” will never pass this test
         v.dist := v.alt
         v.previous := u            // Set the shortest path ”back to source”
      fi
   od
od
```

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

- Backtracking / depth-first, LIFO / FIFO, branch-and-bound, breadth-first and Dijkstra use only local information when choosing the next step.

- A*-search is simular to Dijkstra, but it uses some global heuristics (”qualified guesses”) to make better choices from Q at each step.

- Much used in AI and knowlegde based systems.

- A*-search (as Dijkstra) 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

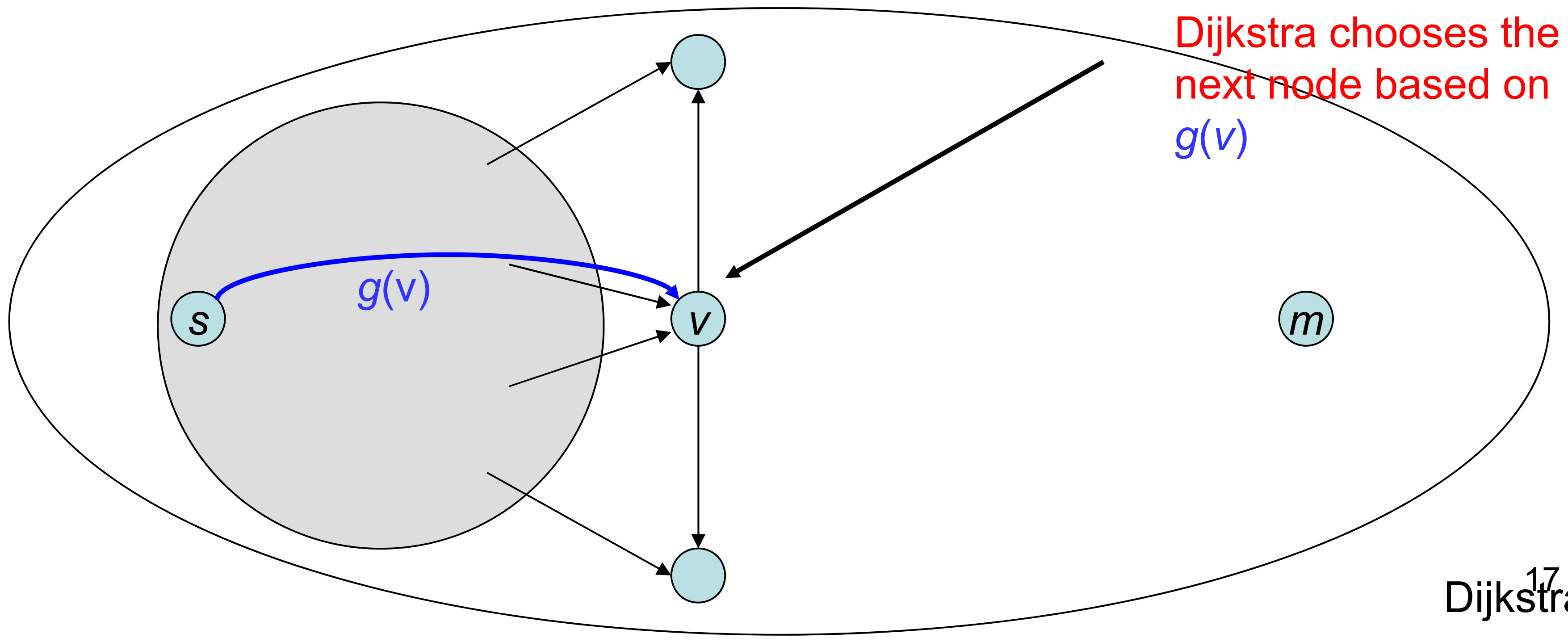
  And (like for Dijkstra) the aim is to find the cheapest (shortest) path from the start node to a goal node

- A*-search: If we for each node in Q can ”guess” how far ut 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 in Dikstra
  - 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 breadth-first search with this value as priority.

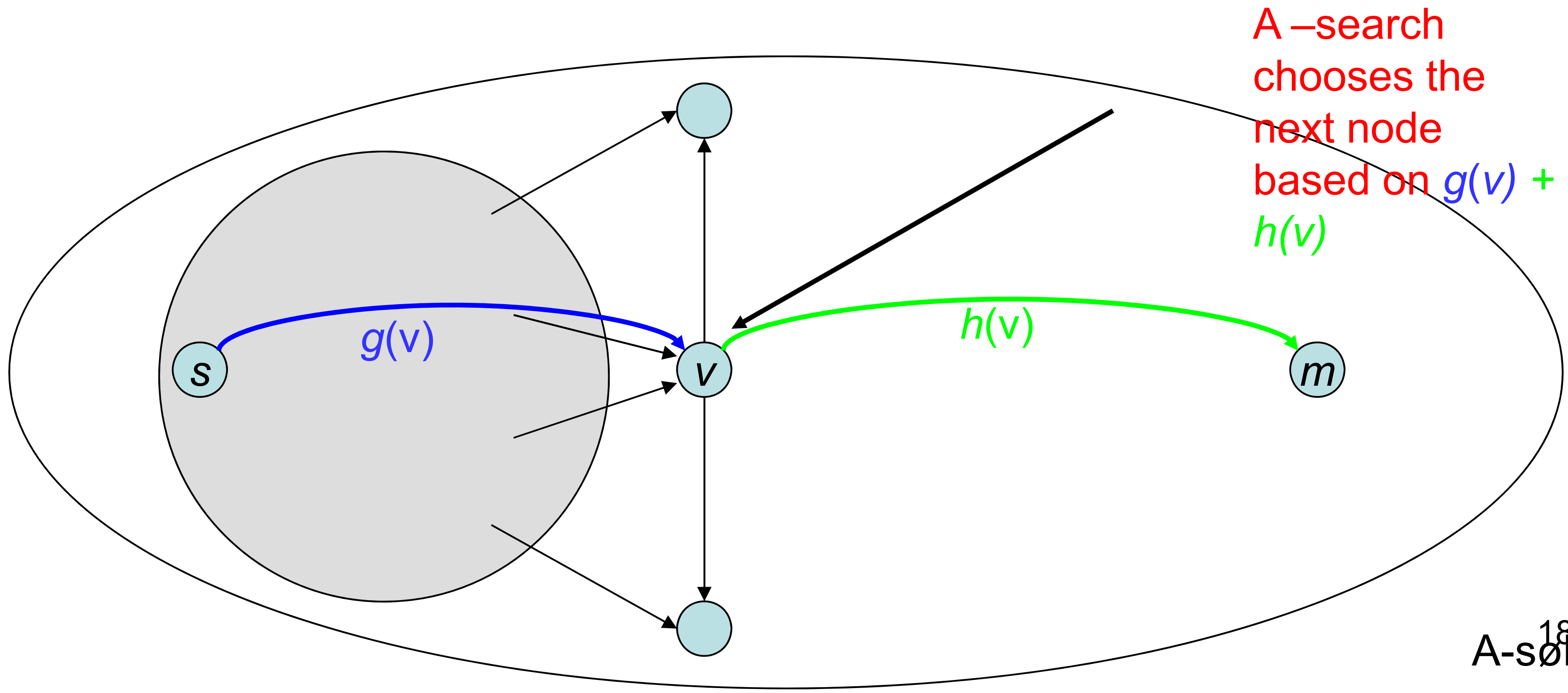


Dijkstra chooses the next node based on *g*(*v*)

A-search – heuristic

- The strategy is a sort of breadth-first search, like in Dikstra
  - 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 breadth-first search with this value as priority.



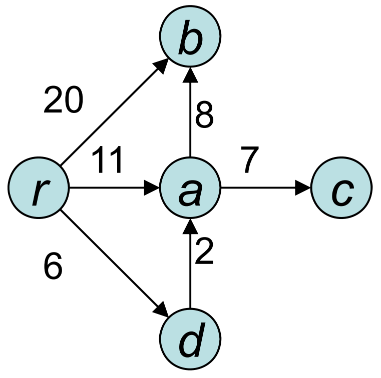
A-search and heuristics

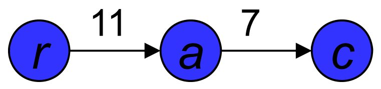
This is in fact already called A*-search in the book

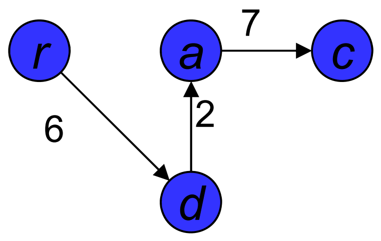
- If we know that *h(v)* is never larger than the really shortest path to the closest goal node
  - and we use the extended Dijkstra algorithm indicated on previous slides, we will finally get the correct result (not proven here)
- However, we will often have to go back to nodes that we «thought we were finished with» (nodes in the tree will have to og back to the priority queue)
  - And this usually gives a lot of extra work
  - Dijkstra never does that and still get the correct answer. We will put requirements on *h(v)* so that this will also be true for the extended algorithm above.
  - Then, we will call it the A*-algorithm

Exercise next week

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







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

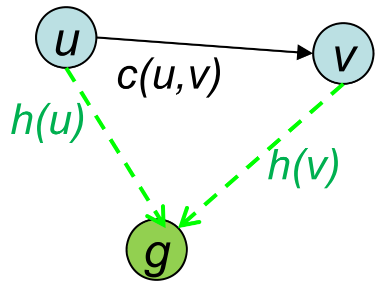
A*-search and heuristics

The function *h(v)* should be an estaimate of the distance from *v* to the closest goal node. However, we can restrict *h*(*v*) further, and get more efficient algorithms.

- To be called an A*-search (not only an A-search, but note that the book names them it a little different), the function *h(v)* must have certain 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:

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



A*-search and heuristics

- To avoid this we add an extra requirement called *monotonicity* on *h*
  - If there is an edge from v to w with weight $c(v,w)$, then we should always have: $h(v) \le c(v,w) + h(w)$
  - Every goal node *m* should have $h(m) = 0$.

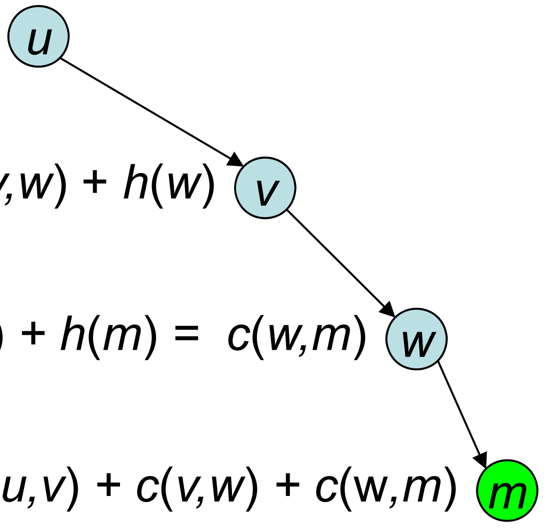
- The nice thing is that if these requirements are fulfilled:
  - then the first requirement (that h(v) is never larger than the real shortest distance) is automatically fulfilled.

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

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

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

$$h(w) \le c(w,m) + h(m) = c(w,m)$$



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

A*-search has monotone heuristics

- 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 can *not* give solutions, so that the algorithm will run faster.
- However, we will then remove the nodes from Q in a differnt order than in Dijkstra
- Thus, we can no longer (without additional proof) know that when *v* is moved from Q to the tree nodes, 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 once more.

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

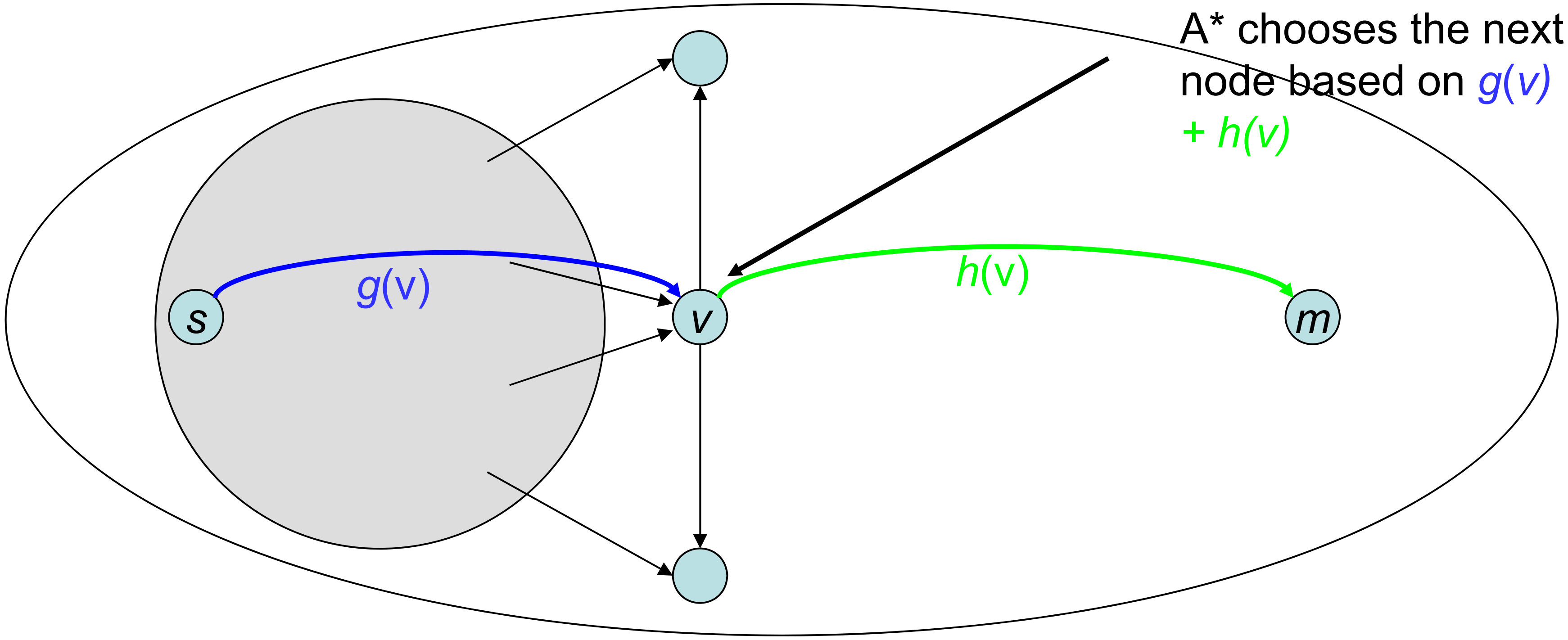
Where: … $h(v) + h(v)$ … appears, it should insted be: … $h(v) \leq g(v) + h(v)$ …

A*-search

- The strategy is a sort of breadth-first search, like in Dikstra
  - 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:

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

- NB: We assume that the heuristic function *h* is monotone



A*-search has monotone heuristics

You will not be asked about details in this argument at the exam, but you should know 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 also for the node v.

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

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

We look at the situation exactly 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}, (which is not v_j) has 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.

A*-search has monotone heuristic

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

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

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 induction step

A*-search – the data for the algorithm

Will be studeied as an exercise next week

- 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*.

- 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 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 int 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)

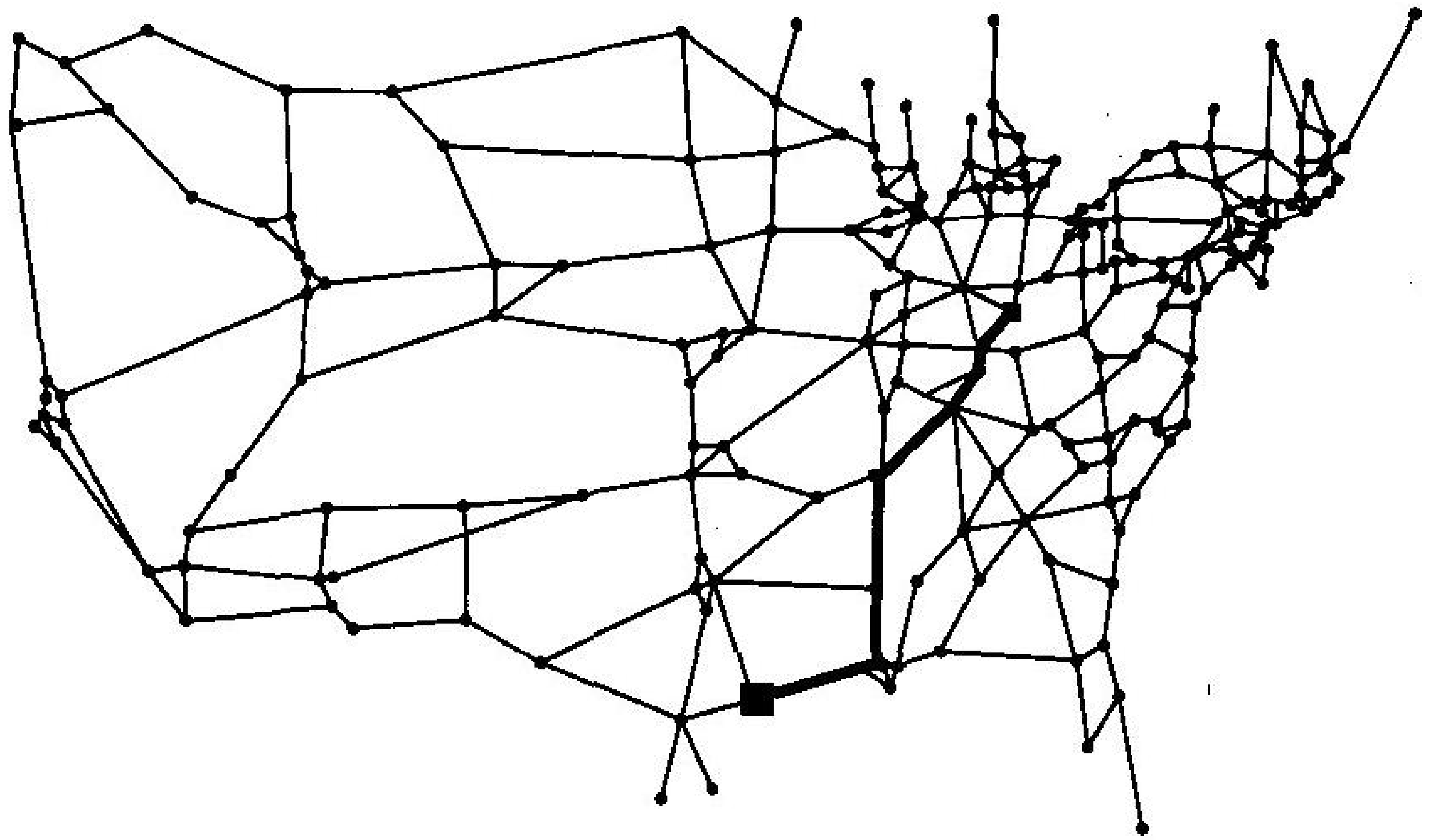
- The main step, that are repeted until Q is empty:

  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.

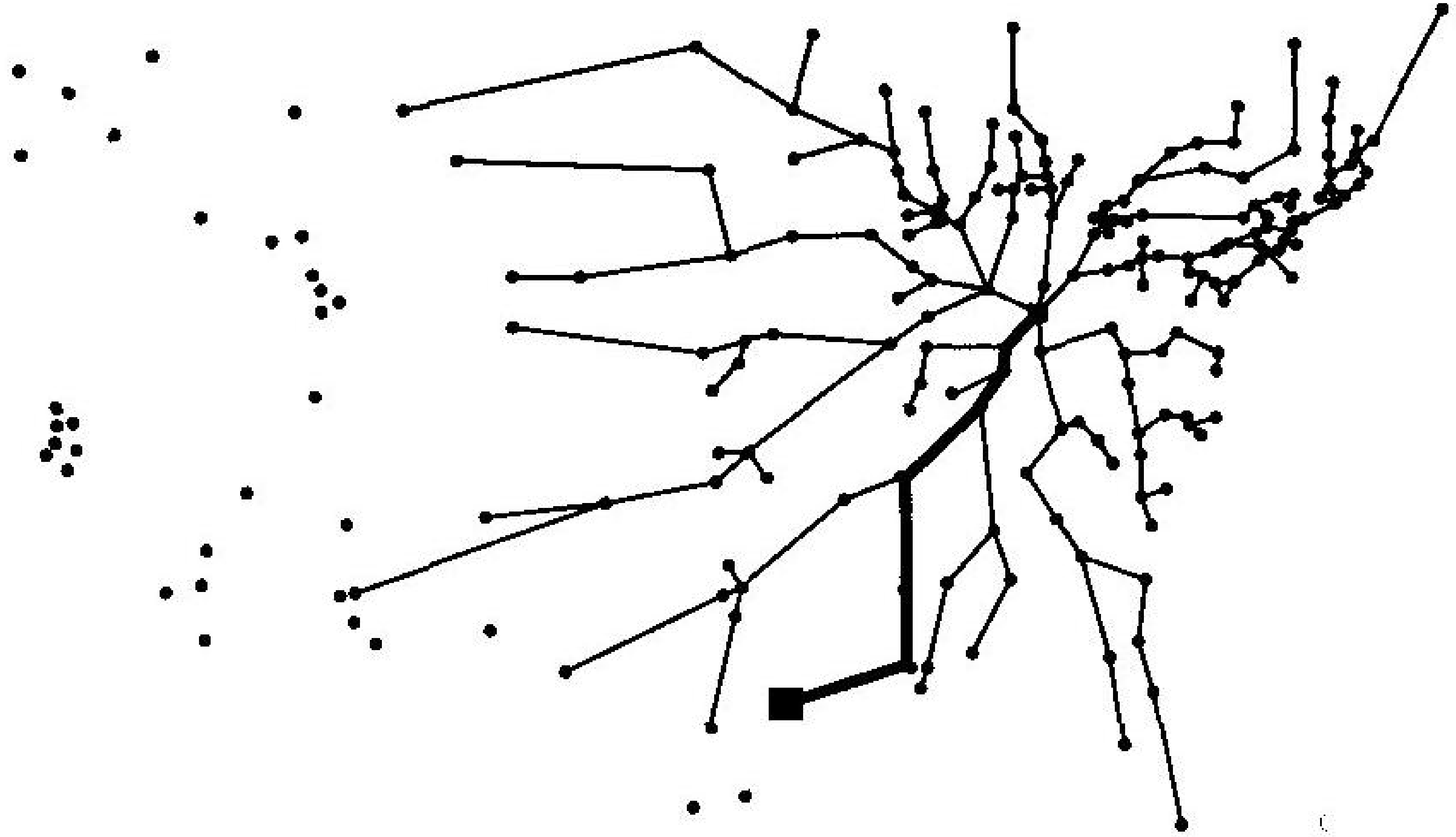
  2 Otherwise, remove v from Q, and let it become a tree node (it now has its parent pointer and g(v) set to correct values)
     - Look at all the unseen neighbours w of v, and for each do the following:
       – Set $g(w) = c(v,w) + g(v)$.
       – Set $f(w) = g(w) + h(w)$.
       – Set $parent(w) = v$.
       – Put *w* into PQ .
     - Look at all neighbours w of v that are in Q, and for each of them do:
       – If $g(w') > c(v,w') + g(v)$ then
         » set $g(w) = c(v,w) + g(v)$
         » set $parent(w) = v$
     - Note that we do *not* look at neighbours of v that are tree nodes. That this will work needs a proof, and it comes on the next slides

A*-search



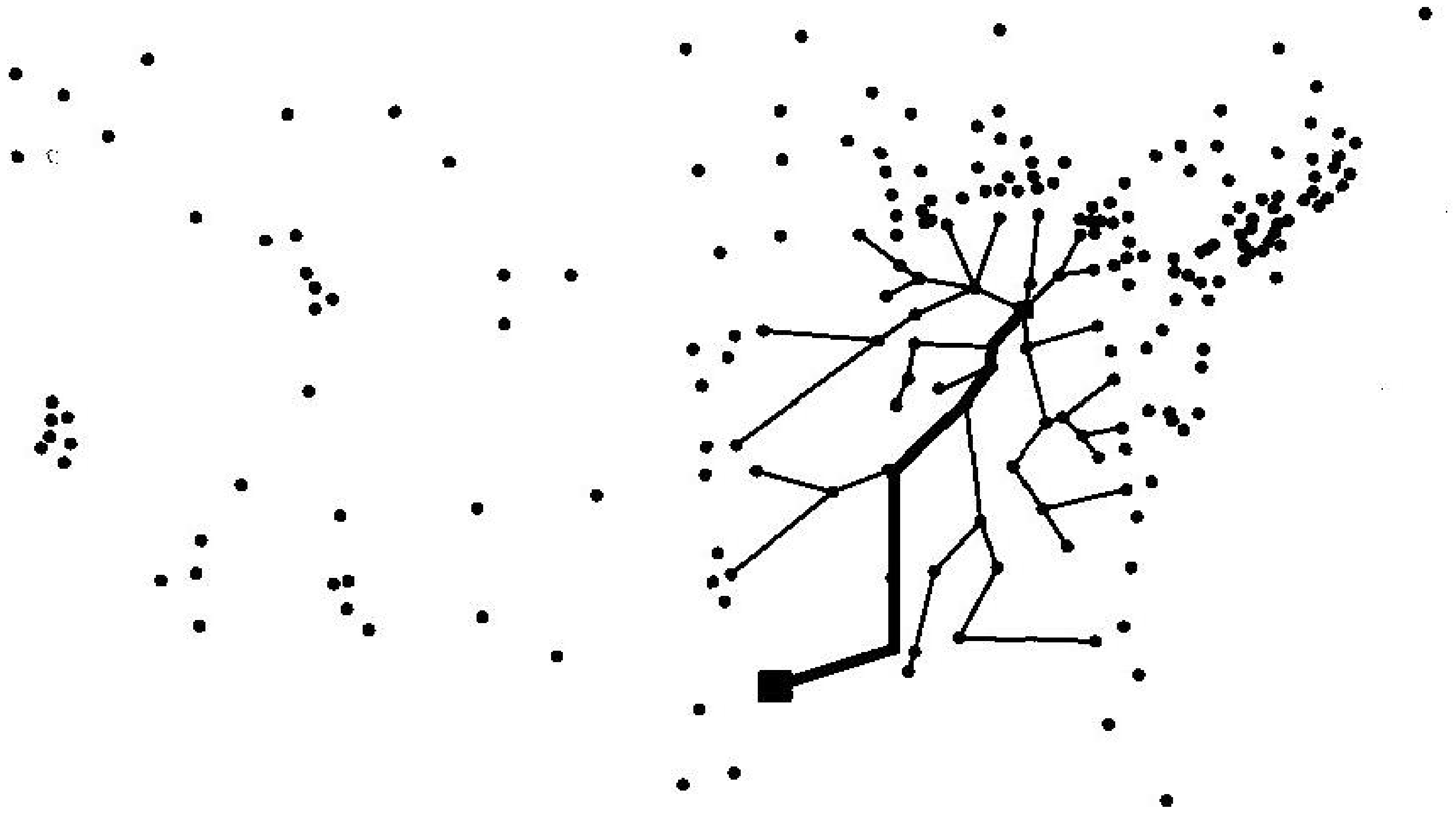
American highways. Shortest path Cincinatti - Houston is marked.

A*-search



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

A*-search



The tree generated by the A*-algorithm with

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