Midterm

Evan will email about that after the lecture, at least 2 lectures from now.

The exam will be given in a regular PDF (not an online form). We will have ~1 week window to work on the exam (and we are requested to not spend more than ~3 hours).

Assignment 1

We have ~2 weeks to complete the assignment. The written problems are to be submitted in PDF (no Latex source is required…).

For the LISP coding assignment: we need to add some functions to the given file.

Side note: Scheme is more restrictive than LISP; a functional language with not a lot of usable libraries, that's why we're using LISP.

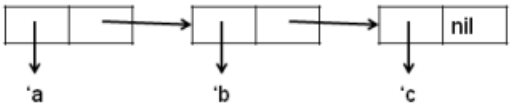
(Short) Intro to LISP

Side note: read the assigned LISP reading for this week! We are expected to learn what's needed on our own…

- Line breaks don't matter
- Everything is in parenthesis
- PREFIX language, e.g.: `(func arg1 arg2)`
- This is a functional language, rather than a procedural language.
- The basic type is **cons cell**: In LISP everything is a list object. For instance: `(cons `a nil)` corresponds to:

  In c it would be something like struct with two fields: car and cdr; the ` is called a quote in LISP, and means: treat the following is a literal (don't evaluate it like a variable / function).
- Nil is used for false, t for true.
- More examples:

  `(cons `a (cons`b (cons`c nil)))` is:



- We can iterate through a list using car and cdr: `car(cons `a nil)` == a, `cdr(cons `a nil)` == nil
- Try not to use variable assignments. For instance, instead of looping we can use recursion.

Today's lecture: Search

What is search?

Whenever we have several immediate options and we need to choose which path to choose.

A classic example as described in the book is path planning: starting at point A, with the goal of getting to point B, we need to figure out the sequence of actions to achieve the path.

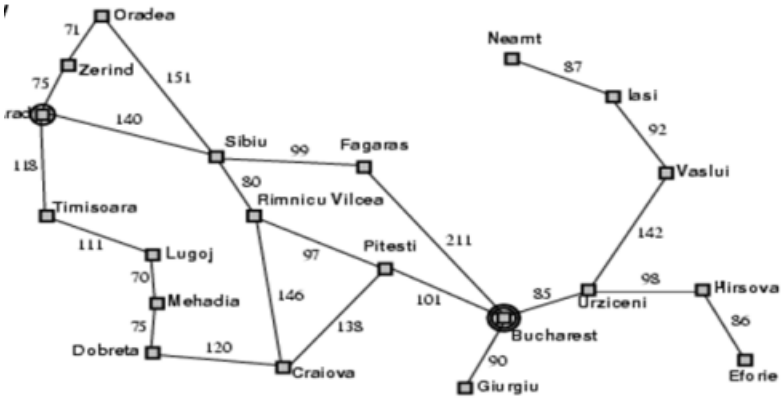
More examples include games, like the 8-puzzle (which we'll talk about later). If there are multiple players it complicates the problem.

When search works

Search works well in:

- Static environments.
- Observable: we have global available information on what we need to know.
- Discrete: a finite number of possible values to the problem's variables. There are non-discrete problems like robotic arm movement, but then the search space is very large.
- Deterministic: we know what the result from each action will be before we take it. For instance, if we take a right turn in some road, we know the car will turn right…

Tree Search



Example search problem: how to get from Arad to Bucharest. Or:

- Initial state: we're in Arad
- Successor function: returns the accessible states from the current state
- Goal test: here – get to Bucharest
- Path cost: in this context it will be time – how long does a path take to travel

So the initial state is **in(Arad)**, the successor function is a pair <action, successor> - the action which we take and the state we get to, like **(Go(Zerind), In(Zerind))**.

Exercise

Given 12 gal, 8 gal and 3 gal jugs, we need to measure 1 gal.

- Initial state: `empty(12g), empty(8g), empty(3g)`
- Goal test: `sum(12g,8g,3g) == 1`
- Successor function: pouring from one jug to another/floor, filling a jug from the faucet…
- Cost function: number of ways, like amount of wasted water on the floor, amount of water used in the entire process.

States vs. Nodes:

GPS Example:

- a state would be (x,y) coordinates;
- a node would be: the current state, what was your previous state, the action you took, path cost, depth.
- A state can have multiple nodes: a state is just your location, but there can be different paths to get there – different nodes.

Applications

- Path planning
- Robot manipulation
- Integrated circuit design – figure out the best layout for chip design

…

The Mother of All Search Algorithms

This is the basic pseudo code that is the base in which we can plug in different algorithms.

- Current_node ← start_node
- Loop:
    - Is current_node = goal_state? If yes, output path and terminate
    - Generate all next states of current_node and add to fringe_set
    - Current_node ← choose from fringe_set

The **fringe set** is the set of all successor nodes that we can move to from current_node.

Depending on how we implement the fringe set affects the way the algorithm works.

Complications

How to choose next state from fringe set? It may have significant implications on

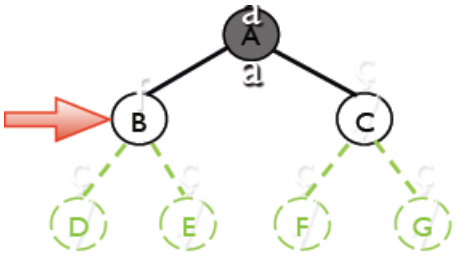
- Time complexity
- Space complexity
- Completeness (does it terminate always?)
- Optimality – the quality of the solution path

Measuring complexity:

- **b**: the maximum branching factor – largest number of next states of any state.
  In our example: the maximum degree of any city in the map (4 in our case)
- **d**: number of steps to the goal – depth of the goal.
- **m**: maximum possible path length, which is not always finite
  In out example $m = \infty$ as we can jump back and forth between any two cities as long as we want.

BFS (Breadth-first search)

- expand shallowest unexpanded node
- Implementation: FIFO – queue for the fringe set.



For instance, in the structure in the figure we first expand a, then b and c, then d, e, f and g.

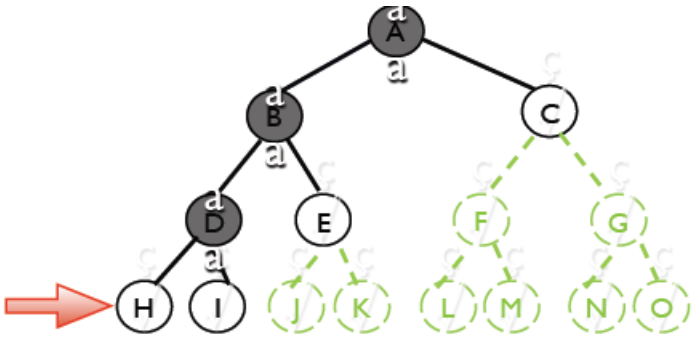
That is, expand row by row.

Properties of BFS

- Completeness: It's never possible for BFS to expand to a lower depth of a previously expanded node, i.e. the path length is monotonically increasing. In short: yes, it is complete.
- Optimal: depends on what exactly is optimal – it's not guaranteed to return the solution of lowest path cost, but it is guaranteed to return the solution with the least amount of steps (cost may be ≠ step). For instance: we may have to go through only 2 cities on our way to our destination, but the path will take 100 km, as opposed to some path through 3 cities that costs only 50 km.
- Space complexity: the size of the fringe set is $O(b^{d+1})$ – since at depth d we have b^d, the total sum is $O(b^{d+1})$

DFS (Depth first Search)

- Expand deepest unexpanded node
- Implementation: LIFO – stack for the fringe set



For instance in the figure: we expand "downwards" first: $A \to B \to D \to H$

When is DFS preferable over BFS?

When going downwards the only nodes in memory take $O(d)$ – because we store only those in the path (in some linear factor). At each step we're adding branching factor b nodes to the fringe, and we're doing that m times (the search ends when $d = m$).

- Space: $O(bm + 1)$ – better than BFS
- Time: (I think it's the same as BFS)
- Complete: **no!** we can get stuck in a loop over some path (e.g.: H leads to A again).

Consider the case where the successor function take us half way to the goal – then we converge but never get to the goal.

Another issue: our goal may be in depth m, but we may start by iterating over another branch that may have infinite depth.

To solve this issue: iterative deepening search

Iterative Deepening Search

- Have a cut-off limit L – you start DFS but halt the deepening on L.
- If finished without finding the goal – increase L and repeat.
- The penalty: every node at depth d is visited $m - d$ times, i.e. m times for the first node, $m - 1$ for the seconds level nodes,… , 1 time for the mth level nodes.

Efficiency

When $L = d$ the number of nodes we're searching is:

$N = b^0 + b^1 + b^2 + \cdots + b^{d-2} + b^{d-1} + b^d$

So the total number of nodes in iterated deepening search to $L = d$:

$$\sum_{d=1}^{m} N_d = (d+1)b^0 + db^1 + \cdots + 3b^{d-2} + 2b^{d-1} + b^d$$

The overhead is not that much…

Bi-directional Search

- As the search deepens, the efficiency declines.
- We minimize depth by applying 2 searches:
    - Forward from the start state

- Backward from the goal

- Catch: could be hard going back from the goal…

What if not all paths are equal?

Uniform-cost Search

- Expand least-cost unexpanded node
- Implementation – the fringe is a priority queue ordered by path cost.
- If all costs are equal, this is the same as BFS

For instance in the figure: the iteration order will be:

Push A -> pop A -> push B,C

Pop B -> push D, E

Pop C -> push F (top), G (bottom)

Pop F, D, E, G

Notation: $g(n)$ is the total cost from start to node n; the fringe is a priority queue ordered by $g(n)$.

Properties

- Completeness: only when all costs have some $\epsilon > 0$ cost
- Time and Space: if the cost for optimal solution is C^* then the cost is $O\left(b^{1+\left\lfloor \frac{C^*}{E} \right\rfloor}\right)$

Graph Search

We keep track of states we've seen before in the closed set (initialized to be empty), this way we remove repeated states.

Summary

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

Typo: floor, not ceiling

Heuristics

- The search problems can be applied on NP-complete problems , so using heuristics we can get good average run-time or good (not optimal necessarily) solution.

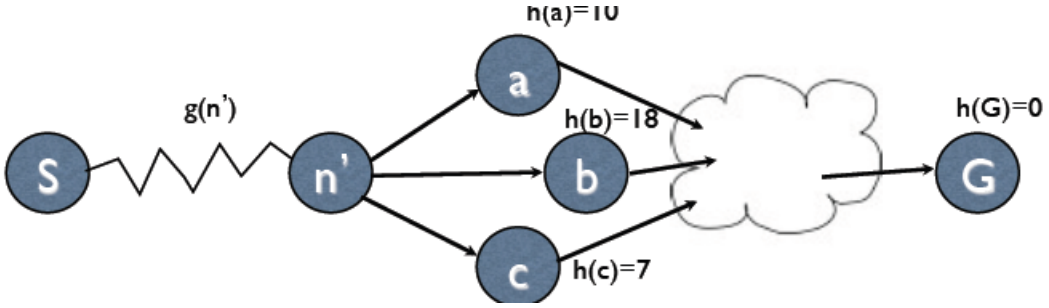
In search

Heuristic in this context is a function that maps a state onto an estimate of the cost to the goal from that state.

Define $h(n)$ as an estimated cost of path from node n to the goal, and for n being the goal $h(n) = 0$.

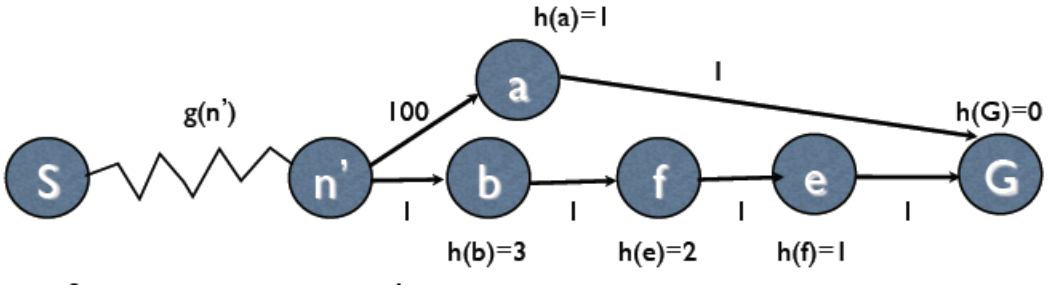
Using Heuristics in informed Search

We can use it to prioritize the fringe. For instance:



We will expand c first like in a cost-search but only using the h values.

But here:

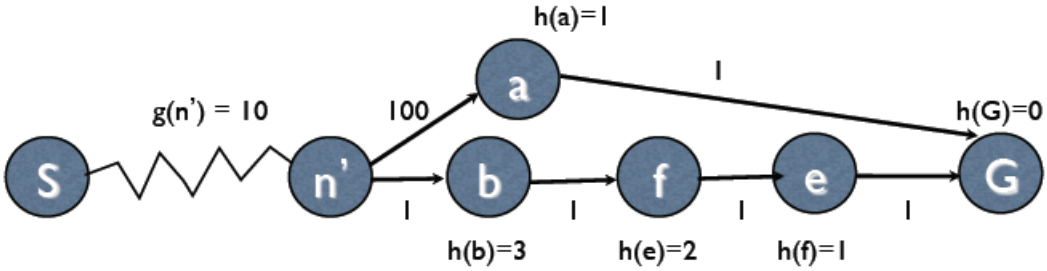


We will choose a over b and then we will pay $g(n') + 101$ instead of $g(n') + 4$

How can that be fixed

We can combine $g(n)$ and $h(n)$:

The fringe priority queue is ordered by: $f(n) = g(n) + h(n)$ – use g for the path we know and h for that we don't know.



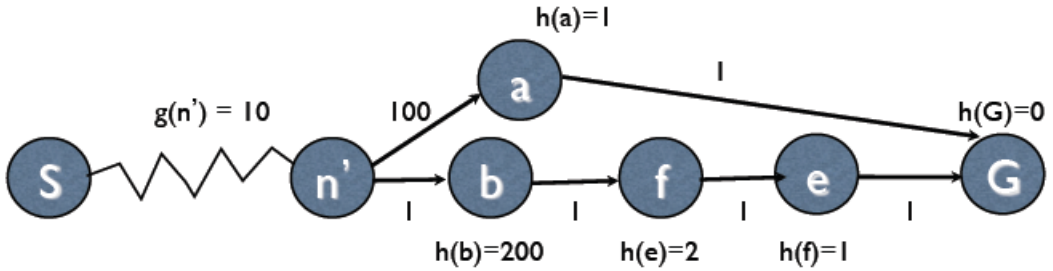
In this case:

$f(a) = 10 + 100 + 1 = 111$

And that's the A^* algorithm.

Is A^* guaranteed optimal

No, due to possible heuristics over-estimates. Like:

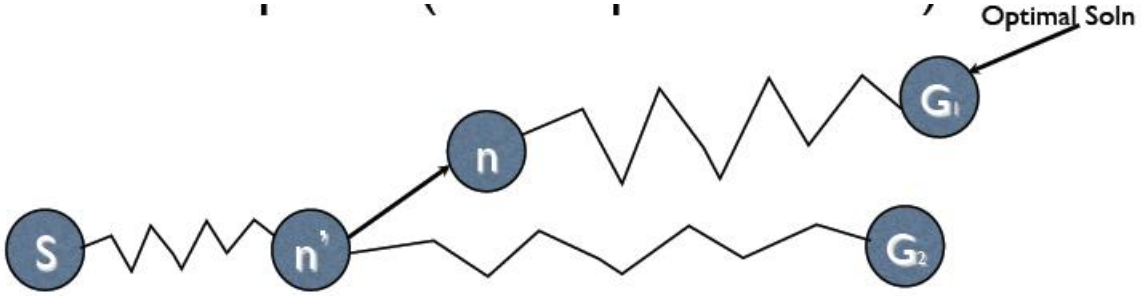


But if we use an admissible ("good") heuristics, we will get a good solution. Let $h^*(n)$ be be the cheapest cost of a path from n to goal. An admissible heuristics is defined as one that for all n: $h(n) \leq h^*(n)$. Examples:

- $h(b) = 4)$ is not admissible, because the real path is actually 3.

Theorem: if $h(n)$ is admissible, A^* is optimal (if no states are repeated)

Proof:

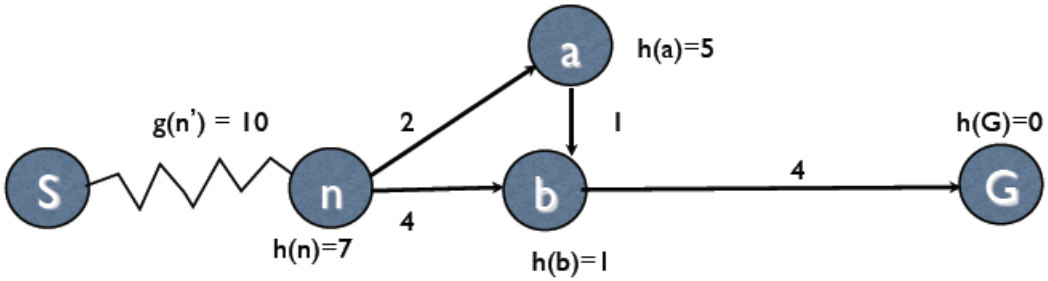


Suppose G_2 is suboptimal and path $(S, \dots, n', \dots, G_2)$ is returned by A^*.

If n is the first unexpanded node in the optimal path to G_1, in A^* the only way G_2 is expanded before n is if $f(n) > f(G_2)$, which means $f(G_1) \geq f(G_2)$ (since $f(G_1) \geq f(n)$)– contradiction, since the initial assumption is that G_1 is the optimal solution.

This may not be correct – should go through the proof in the slides.

When Admissibility Fails



…

We also need to ensure **consistency** – the triangle inequality. That is we require:

$h(a) \leq h(b) + cost(a, b)$.

That's why the heuristic in the previous figure is admissible but inconsistent.

Consistent heuristics:

Iff for every node n and every successor n' of n generated by any action a then: $cost(n, a, n') + h(n') \geq h(n)$

This simply means that the δ in the f-cost needs to be nonnegative, so f is non-decreasing along any path.

Consistency implies admissibility

Proof by induction:

Base: $h^*(n) = 0$; n is the goal state so $h(n) = 0$ by definition for heuristic on goal state.

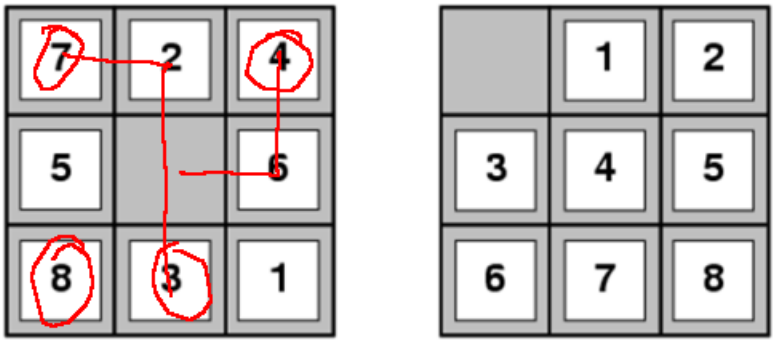
Inductive: assume for all $n' > n$ on shortest path to goal: $h(n') \leq h^*(n')$

Lemma: $h(n) \leq h^*(n)$

The rest is in the slides.

Properties of A^*

- Completeness: yes
- Exponential time complexity
- Space: all nodes in memory
- Optimal: with consistent heuristic and graph-search



Manhattan distance is a good heuristic – admissible. For instance: $h(7) = 3$ (like the "real" distance from the current state to the goal). Another admissible heuristic: number of tiles that are misplaced.

Dominance

If for all n $h_2(n) \geq h_1(n)$ and h_1, h_2 are admissible, then h_2 **dominates** h_1. In this case Manhattan distance dominates.

Simplified memory-bounded A^*

In the slides

Iterative deepening A^* (IDA^*)

In the slides...

Recursive Best First Search (RBFS)

Keep track of the best alternative path – f-limit.

When current f-values exceed f-limit, backtrack and store best f-values of children at backtracked nodes.