Lecture 04

Search Algorithms

'Uninformed' and 'Informed' Search

Jeremy Pitt
Department of Electrical & Electronic Engineering
Imperial College London, UK
Email: j.pitt _at_ imperial _dot_ ac _dot_ uk
Phone: (int) 46318
Office: 1010 EEE

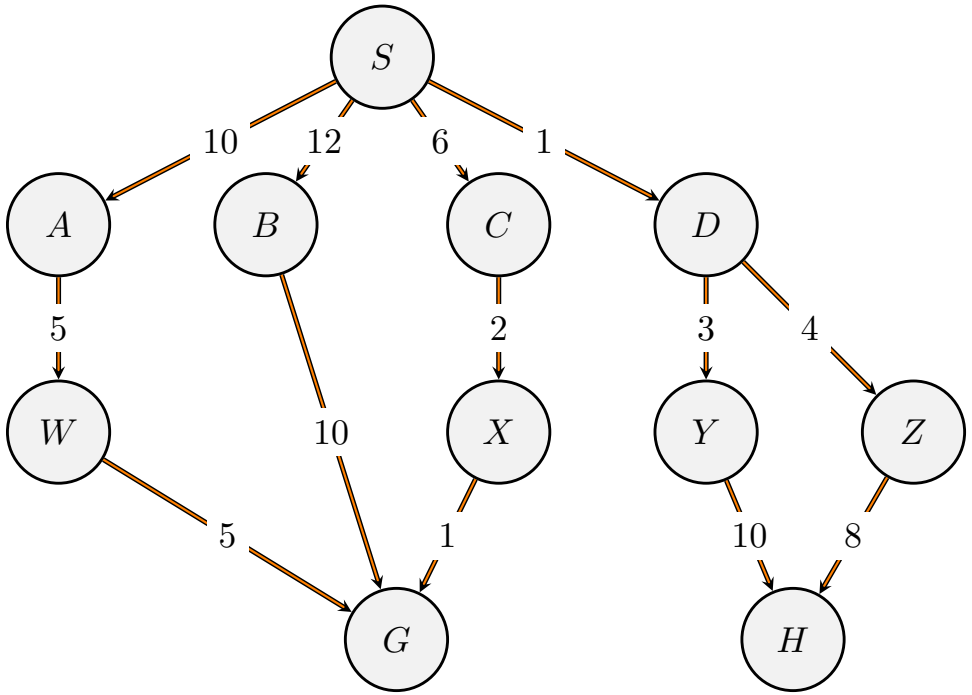
Aims and Objectives

- Aims
  - To study graph-search algorithms which search a graph for a solution path in a systematic, but 'uninformed' fashion
  - To study graph-search algorithms which search a graph for a solution path in a systematic, but a less 'uninformed' fashion
  - To describe a framework with different criteria for comparing and contrasting different algorithms

- Objectives
  - Understand how to instantiate the GGSE with different search algorithms
  - AI Problem Solver = Search-Space-Formulation + GGSE + algorithm

Example: Route Finding

- Simple route finding example
  - Paths are uni-directional
  - Paths have weights ('cost' to traverse them)
  - Requirement is to find a path from S to some other node G



Agenda

- We will study how to complete the General Graph Search Engine for different algorithms
  - Uninformed Search
    - Breadth First
    - Depth First
    - Uniform Cost
    - Iterative Deepening Depth First
    - Beam
  - Informed Search
    - Best First
    - A*
- Compare and contrast with respect to evaluation criteria
- Note capital letter node names in the example need to be converted to lower-case Prolog atoms for a completion with the GGS

GGSE: Recalled

```
search( Graph, [Node|Path] ) :-
    choose( [Node|Path], Graph, _ ),
    state_of( Node, State ),
    goal_state( State ).
search( Graph, SolnPath ) :-
    choose( Path, Graph, OtherPaths ),
    one_step_extensions( Path, NewPaths ),
    add_to_paths( NewPaths, OtherPaths, GraphPlus ),
    search( GraphPlus, SolnPath ).
```

- Query the GGSE with what we know – the sub-path of the graph (of the search space) with just the initial state
  - `?- search( [ [start_state] ], SolnPath ).`
  - In the example: `?- search( [ [s] ], SolnPath ).`
- `search` can **fail**: there may not be a solution path

One Step Extensions

- Computing all the nodes reachable by applying the state change rules to a node

```
– one_step_extensions( [Node|Path], NewPaths ) :-
      state_of( Node, State ),
      findall( [NewNode,Node|Path],
        ( state_change( Rule, State, NewState ),
          make_node( Rule, NewState, NewNode ) ),
        NewPaths ).
```

- Now to complete GGSE we have to define the behaviour of
  - `choose/3`
  - `add_to_paths/3`
- Different 'completions' will give us different algorithms with different properties, as per the evaluation criteria

Evaluation Criteria

- Different search algorithms can be evaluated and compared in terms of 3 criteria
  - Completeness – is the algorithm guaranteed to find a solution, assuming there is one?
  - Optimality – does the algorithm find the highest quality solution when there is more than one to choose from?
  - Complexity
    - Time: how long does the algorithm take to find a solution?
    - Space: how much memory does it need to perform the search?

- The choice of an appropriate search method can be made based on matching these criteria against
  - The (expected) type of search space
  - Available resources (space and time)
  - Actual requirements (any —, best —, 'good enough' solution)

Comparing Search Algorithms (1)

- Completeness: binary
  - Results exist for 'standard' algorithms
  - If you invent your own algorithm, you should do a completeness proof
- Optimality: binary
  - Results exist for exhaustive search
  - Introducing heuristics to narrow the search space makes optimality dependent on the choice of the heuristic
  - In general, the more 'efficiently' the space is searched, the harder it is to prove

Comparing Search Algorithms (2)

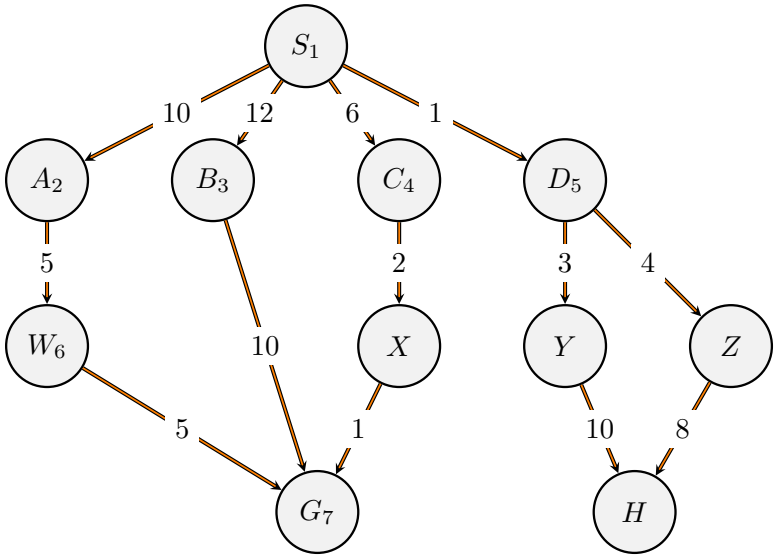
- Complexity: units of time/space of the 'principal activity'
  - Big $\mathcal{O}$ notation: the time(space) complexity of an algorithm A is $\mathcal{O}(f)$ ("of the order of f") means that the actual (worst case) time(space) requirement is bounded by some constant times f
  - Critical factors in complexity of graph search: branching factor b, depth of solution d
    - Branching factor: a hypothetical search space where every state can be expanded to give b new states
    - Solution depth: suppose a solution is found at depth d (i.e. d nodes 'beneath' the root node)
  - Most search algorithm time/space complexity results are $\mathcal{O}$(some function of b and d)

Some Notes on Complexity

- Notes
  - Abstract away from actual number of operations and size of data:
    - Concerned with the number of times it is necessary to **expand** a node (i.e. apply all the state change rules to the node), and the number of nodes to store in memory (irrespective of the actual number of instructions and size of data)
  - In general, complexity is OK if linear or low polynomial ($\mathcal{O}(n)$ or $\mathcal{O}(n^2)$), things look a little bleaker if its exponential ($\mathcal{O}(n!)$)
  - Generally focus on **worst case** results, there is an argument to be had about **average case**
  - Although the numbers for time stack up quickly for time for polynomial and exponential complexity, generally space is the killer
  - Just as computers get faster, they get applied to bigger, more complicated problems with larger input size. So actually design, implementation and use of efficient algorithms are arguably more important, not less.

Breadth First (BF): The Algorithm

- In general
  - Root node expanded first
  - All nodes generated by root expanded next, then their successors, etc.
    - All nodes at depth d are expanded before all those at depth $d+1$, all nodes at d+1 are expanded before all those at depth $d+2$, ...
    - For the example (subscripts denote order of expansion)



BF: Declarative Specification

- Completion

```
- choose( Path, [Path|OtherPaths], OtherPaths ).
- add_to_paths( NewPaths, OtherPaths, AllPaths ) :-
      append( OtherPaths, NewPaths, AllPaths )
```

BF: In Operation

- Given as goal, node G – test and expand:
  - `[ [S] ]`
    $\Longrightarrow$ `[ [A,S], [B,S], [C,S], [D,S] ]`
    $\Longrightarrow$ `[ [B,S], [C,S], [D,S], [W,A,S] ]`
    $\Longrightarrow$ `[ [C,S], [D,S], [W,A,S], [G,B,S], ]`
    $\Longrightarrow$ `[ [D,S], [W,A,S], [G,B,S], [X,C,S] ]`
    $\Longrightarrow$ `[ [W,A,S], [G,B,S], [X,C,S], [Y,D,S], [Z,D,S] ]`
    $\Longrightarrow$ `[ [G,B,S], [X,C,S], [Y,D,S], [Z,D,S], [G,W,A,S] ]`

- Solution path $=$ `[G,B,S]`

BF: Evaluation

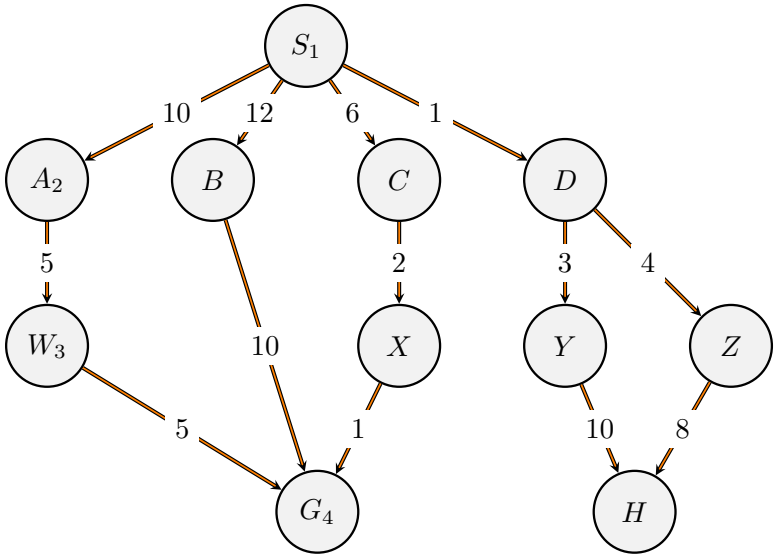
- Completeness
  - Yes: if there is a solution it will be found (exhaustive search)
- Optimality
  - Yes: if **path cost** is a non-decreasing function of node depth
  - Breadth first finds the shortest path to a goal
- Complexity
  - The maximum number of nodes expanded before finding a solution is

$$1 + b + b^2 + b^3 + \ldots + b^d$$

  - Therefore time complexity is $\mathcal{O}(b^d)$
  - All the nodes have to be stored, so space complexity is $\mathcal{O}(b^d)$
  - Exponential complexity has a number of practical implications

Depth First (DF): The Algorithm

- In general
  - Root node expanded first
  - Always expand one of the nodes on the deepest level of the tree
  - If the search hits a 'dead end' (a non-goal state with no expansion), backtrack and try unexpanded node at the next deepest level, etc.



DF: Declarative Specification

- Just change two variables from BF
  - `choose( Path, [Path|OtherPaths], OtherPaths ).`
  - `add_to_paths( NewPaths, OtherPaths, AllPaths ) :-`
    `append( NewPaths, OtherPaths, AllPaths )`

- Notes
  - Loop-checking for search spaces with loops or 'reversible' actions
  - **Depth-limited search**: depth-first search with a limit on the path length (i.e. depth), for search spaces with 'infinite' paths (and loops)
  - Prolog's own search strategy it top-down, depth first
  - Therefore we could just work on one path and use the Prolog engine to 'remember' the choice points and backtrack to them if they are needed (i.e. frontier node of current path has no expansion)

GGSE 'Lite'

- Generate the alternative expansions of a node one by one, by backtracking to the state change rules, rather than computing them all at once using `findall`

- ```
  search( [Node|Path], [Node|Path] ) :-
      state_of( Node, State),
      goal_state( State ).
  search( [Node|Path], SolnPath ) :-
      state_of( Node, State ),
      state_change( Rule, State, NewState ),
      make_node( Rule, NewState, NewNode ),
      search( [NewNode, Node|Path], SolnPath ).
  ```
  ```

DF: In Operation

- Given as goal, node G – GGSE 'original' – test and expand
  - `[ [S] ]`
    $\Longrightarrow$ `[ [A,S], [B,S], [C,S], [D,S] ]`
    $\Longrightarrow$ `[ [W,A,S], [B,S], [C,S], [D,S] ]`
    $\Longrightarrow$ `[ [G,W,A,S], [B,S], [C,S], [D,S] ]`
- Solution path = `[G,W,A,S]`
- GGSE 'lite'
  - `[S]`
    $\Longrightarrow$ `[A,S]`
    $\Longrightarrow$ `[W,A,S]`
    $\Longrightarrow$ `[G,W,A,S]`
- Solution path = `[G,W,A,S]`

DF: Evaluation

- Completeness
  - No: search spaces with infinite paths or loops
- Optimality
  - No: returns the first solution that is found
- Complexity
  - 'original', with m being the maximum depth of the search tree
    - Time $\mathcal{O}(b^m)$; Space $\mathcal{O}(b * m)$
  - 'original', with l being the limit on the path length
    - Time $\mathcal{O}(b^l)$; Space $\mathcal{O}(b * l)$
  - 'lite', with m being the maximum depth of the search tree
    - Time $\mathcal{O}(b^m)$; Space $\mathcal{O}(m)$

Aside

- Recall the inductive definition of paths in a graph defined $G' = \langle S, Op \rangle$

$$P_{G'} = \bigcup_{i=0}^{\infty} P'_i = P'_0 \cup P'_1 \cup P'_2 \cup \ldots$$

- Breadth First searches all the paths in P'_0 before searching any in P'_1, searches all the paths in P'_1 before searching any in P'_2, and so on

- Depth First searches *one* of the paths in P'_i, where i is the deepest level of the search; if there are no more paths, it tries *one* of the paths in P'_{i-1}, and so on

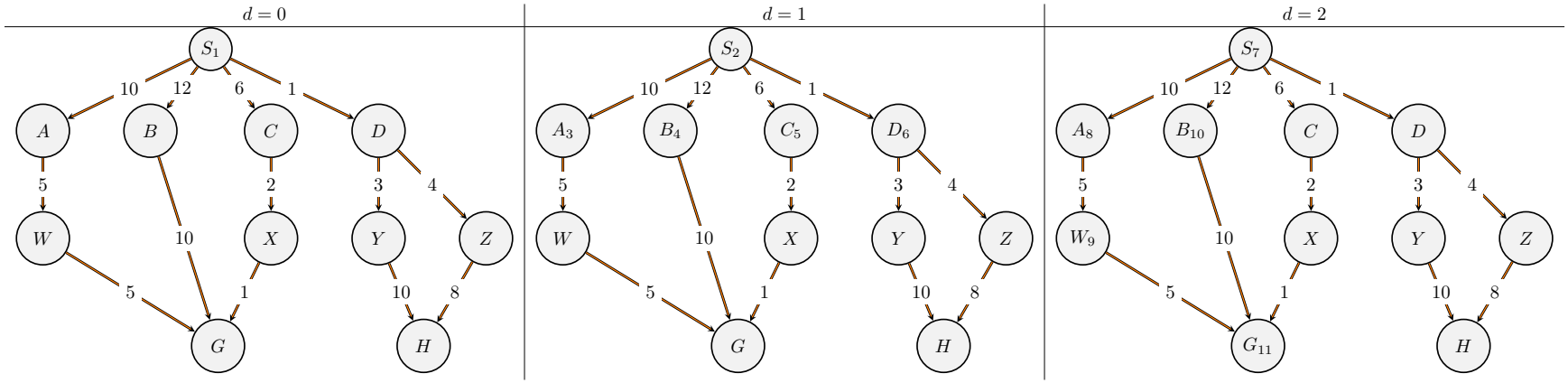
- Note that swapping BF for DF exchanges completeness and optimality for linearity in space complexity

- Still got exponential time complexity

Iterative-Deepening Depth First (IDDF): The Algorithm

- Avoid the pitfalls of depth-first search by imposed a limit on the path length (depth-limited search)

- IDDF is depth-limited search at successive increments of the limit
  - First do depth-limited search with $l = 0$, then with $l = 1$, then with $l = 2$, and so on



IDDF: Declarative Specification

- Add a depth argument to `search/3` (for all heads and recursive calls)

```
search( +Graph, ?SolnPath, +Depth )
```

- Add a new clause to `search/3` (make it the *second* case)

```
search( Graph, SolnPath, Depth ) :-
  choose( Path, Graph, GraphMinus ),
  pathdepth( Path, PathDepth ), %depth is length-1
  PathDepth = Depth, !, %Note the 'cut' here
  search( GraphMinus, SolnPath, Depth ).
```

- Add a 'wrapper' `id_search/3` for search (recurse on `id_search`!)

```
id_search( Paths, SolnPath, Depth ) :-
  search( Paths, SolnPath, Depth ).
id_search( Paths, SolnPath, Depth ) :-
  Depth1 is Depth + 1,
  id_search( Paths, SolnPath, Depth1 ).
```

IDDF: In Operation

- Initial query: `id_search( [ [start_state] ], SolnPath, 0 ).`
- Given as goal, node G – test (goal), test (path length) and expand
- limit=0
  - `[ [S] ]`
    $\Longrightarrow$ `[ ]`
    $\Longrightarrow$ `fail`
- limit=1
  - `[ [S] ]`
    $\Longrightarrow$ `[ [A,S], [B,S], [C,S], [D,S] ]`
    $\Longrightarrow$ `[ [B,S], [C,S], [D,S] ]`
    $\Longrightarrow$ `[ [C,S], [D,S] ]`
    $\Longrightarrow$ `[ [D,S] ]`

$\Longrightarrow$ `[ ]`
$\Longrightarrow$ `fail`

- limit=2
  - `[ [S] ]`
    $\Longrightarrow$ `[ [A,S], [B,S], [C,S], [D,S] ]`
    $\Longrightarrow$ `[ [W,A,S], [B,S], [C,S], [D,S] ]`
    $\Longrightarrow$ `[ [B,S], [C,S], [D,S] ]`
    $\Longrightarrow$ `[ [G,B,S], [C,S], [D,S] ]`
    $\Longrightarrow$ `[ ]`

- Solution path = `[G,B,S]` (same as beadth first, but we used depth first)

- In general, for a solution at depth d, the nodes at depth d are tried once, those at depth $d-1$ twice, and so on up to the root note (at depth 0) which is tried $d+1$ times

IDDF: Evaluation

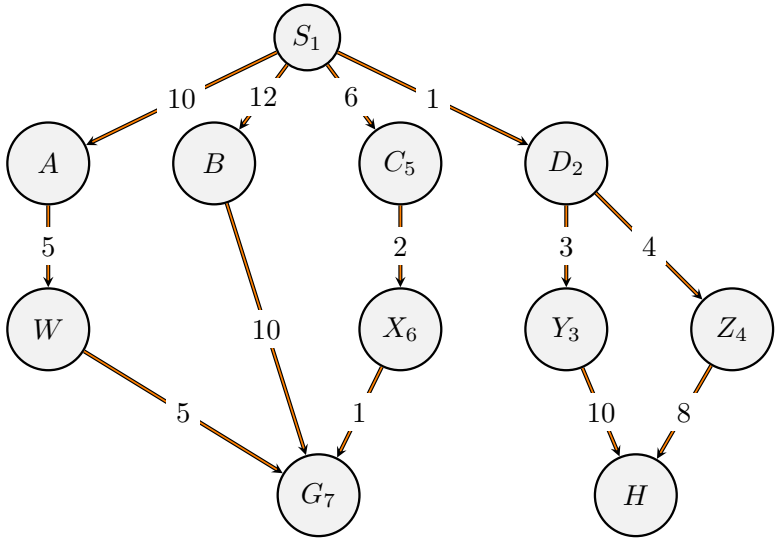
- Completeness
  - Yes: depth-limited search is exhaustive
- Optimality
  - Yes: on condition that the shortest path length is 'best' solution
- Complexity
  - The maximum number of nodes expanded is

$$(d+1).1 + (d).b + (d-1).b^2 + \ldots + (2).b^{d-1} + (1).b^d$$

  - Time complexity is still $\mathcal{O}(b^d)$
  - Space complexity is $\mathcal{O}(b * d)$ (same as depth-first search with $m = d$)
  - Completeness/optimality of BF with time/space complexity of DF

Uniform Cost (UC): The Algorithm

- Expand the path with the lowest cost node on the search frontier
- As calculated by some **path cost function** g



UC: Declarative Specification

- Node representation includes both state and accumulated cost, e.g. (a,0)
- State change rule has to implement path cost function g

```
state_change( Rule, State, NewState, GCost ) :-
  ..., g(..., GCost), ...
```

  - Aside: this is the edge value in the incidence relation

- One step extension is slightly more complex:

```
one_step_extensions( [Node|Path], NewPaths ) :-
    state_of( Node, State ),
    cost_of( Node, PathCost ),
    findall( [NewNode,Node|Path],
      ( state_change( Rule, State, NewState, GCost ),
        AccCost is GCost + PathCost,
        make_node( Rule, NewState, AccCost, NewNode ) ),
      NewPaths ).
```

- Either:
  - `choose` picks the path with the lowest path cost, OR
  - insert paths in increasing order of path cost in `add_to_paths`

UC: In Operation

- Given as goal, node G, order the paths in increasing g-cost of frontier node, choose the first

  - `[ [(S,0)] ]`
    $\Longrightarrow$ `[ [(D,1),(S,0)], [(C,6),(S,0)], [(A,10),(S,0)], [(B,12),(S,0)] ]`
    $\Longrightarrow$ `[ [(Y,4),(D,1),(S,0)], [(Z,5),(D,1),(S,0)],`
    `[(C,6),(S,0)], [(A,10),(S,0)], [(B,12),(S,0)] ]`
    $\Longrightarrow$ `[ [(Z,5),(D,1),(S,0)], [(C,6),(S,0)],`
    `[(A,10),(S,0)], [(B,12),(S,0)], [(H,14),(Y,4),(D,1),(S,0)] ]`
    $\Longrightarrow$ `[ [(C,6),(S,0)], [(A,10),(S,0)], [(B,12),(S,0)],`
    `[(H,13),(Z,5),(D,1),(S,0)], [(H,14),(Y,4),(D,1),(S,0)] ]`
    $\Longrightarrow$ `[ [(X,8),(C,6),(S,0)], [(A,10),(S,0)], [(B,12),(S,0)],`
    `[(H,13),(Z,5),(D,1),(S,0)], [(H,14),(Y,4),(D,1),(S,0)] ]`
    $\Longrightarrow$ `[ [(G,9),(X,8),(C,6),(S,0)], [(A,10),(S,0)], [(B,12),(S,0)],`
    `[(H,13),(Z,5),(D,1),(S,0)], [(H,14),(Y,4),(D,1),(S,0)] ]`

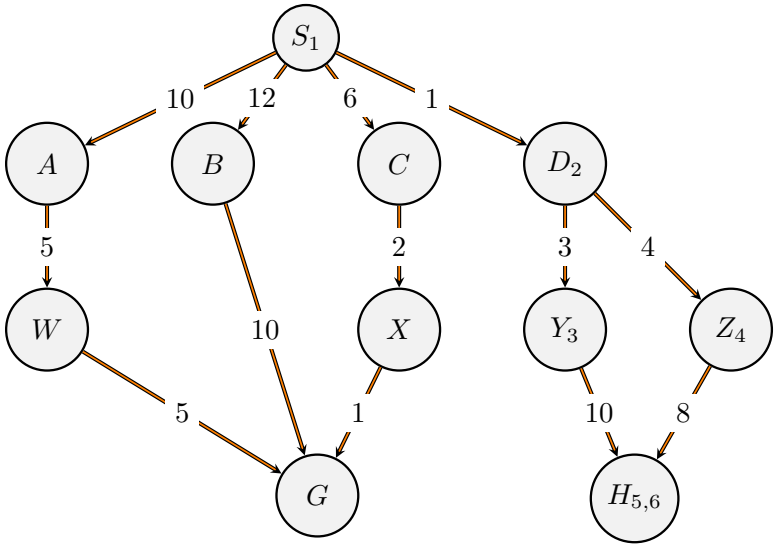
- Solution path = `[G,X,C,S]`

UC: Evaluation

- Completeness
  - Yes
- Optimality
  - Yes: on condition that the cost of getting to a successor of any node is equal to or greater than the cost of getting to that node
- Complexity
  - Breadth-first search is uniform-cost search with $\forall n.g(n) = depth(n)$
  - Time complexity is still $\mathcal{O}(b^d)$ (may have to expand all nodes)
  - Space complexity is $\mathcal{O}(b^d)$ (may have to store all nodes)

Beam Search: The Algorithm

- Limit the graph to a pre-determined number of candidate paths
- If the union of new paths and other paths exceeds the limit, then the 'worst' paths (highest cost of getting to frontier node) are discarded



Beam: Declarative Specification

- Pass the beam width as an extra parameter

```
– search( +Paths, ?SolnPath, +Beam )
```

- Prune the Graph in add_to_paths

```
add_to_paths( NewPaths, OtherPaths, GraphMinus, Beam ) :-
  insert_in_order( NewPaths, OtherPaths, AllPaths ),
  prune( AllPaths, Beam, GraphMinus ).

prune( [ ], _, [ ] ).
prune( _, 0, [ ] ) :-
  !.
prune( [H|T1], Beam, [H|T2] ) :-
  NarrowBeam is Beam - 1,
  prune( T1, NarrowBeam, T2 ).
```

Beam: In Operation

- Beam width = 2 (rather unlikely)

```
    [ [(S,0)] ]
⟹  [ [(D,1),(S,0)], [(C,6),(S,0)] ]
⟹  [ [(Y,4),(D,1),(S,0)], [(Z,5),(D,1),(S,0)] ]
⟹  [ [(Z,5),(D,1),(S,0)], [(H,14),(Y,4),(D,1),(S,0)] ]
⟹  [ [(H,13),(Z,5),(D,1),(S,0)], [(H,14),(Y,4),(D,1),(S,0)]]
⟹  [ [(H,14),(Y,4),(D,1),(S,0)] ]
⟹  [ ]
```

- Solution Path = fail

Beam: Evaluation

- Completeness
  - No
- Optimality
  - No
- Complexity (remember: worst case)
  - When it 'really counts', at depth d, only expand beam width w nodes
  - Time complexity is $\mathcal{O}(w * d)$, assuming a solution is found
  - Only keep beam width w paths each with m nodes, where m is the maximum depth of the tree $that\ is\ searched$
  - Space complexity is $\mathcal{O}(w * m)$
  - Complexity is linear at the expense of optimality and completeness
- Application: machine translation

Intermission

- Simple problem
- Five algorithms
- Five different evaluations
- Four different answers (including one answer, "no")
- Can we do better that?
  - Can we get the algorithm itself to do better than that?
  - It is, after all, a course on (a sub-field of) "artificial" "intelligence"
- So let's try to work smarter, not harder

Another Example: Grid Mazes

- A robot at one location wants to find a route from its current location to a goal location

- It can sense its surroundings; turn left and right; move up, down, left or right one location

- Assumptions
  - All actions are executed perfectly
  - The robot 'fits' one grid location
  - The world is an integer number of grids in both dimensions
  - The robot knows its location, the goal, and has 'mapped' the maze

Grid Maze: Search Space Formulation

- State representation: location in terms of x and y coordinates
  - 2-tuple (X,Y)
- Initial (start) state and final (goal) state(s) are coordinates
- State change rules for movements up, down, left and right

```
state_change( up, (X,Y), (X,Y1) ) :-
  Y1 is Y + 1,
  \+ is_wall( (X,Y), (X,Y1) ), %internal walls
  in_y_boundary( Y1 ), %still in maze
```

- Supplementary constraints
  - Walls represented as facts, e.g. `wall( (1,1), (1,2) ).`
  - `is_wall/2` checks there is not a wall between first coordinate and the second, and vice versa
  - `in_y_boundary/1` checks new y coordinate is in bounds

Best First: The Algorithm

- General Idea
  - Uniform cost expands the path with the frontier node with the lowest $actual$ cost from the start state to the (state of the) frontier node
  - We would prefer to expand the frontier node with the lowest cost from its state to a goal state
  - The cost of reaching any state cannot be determined exactly
  - Apply an evaluation function to make an **estimate** of the cost
  - Use a **heuristic**: a function that calculates cost estimates
    - $h(State, HCost)$: $HCost$ is always 0 if $State$ is a goal state

- Algorithm
  - Instead of ordering paths according to increasing GCost, order them according to increasing HCost
  - Always expand the node with the best (estimated) cost to a goal state before any others

BF: Declarative Specification

- Node representation includes both state and estimated cost, e.g. (a,H)

- State change rule has to implement heuristic function h

```
state_change( Rule, State, NewState, HCost ) :-
  ..., h(..., HCost), ...
```

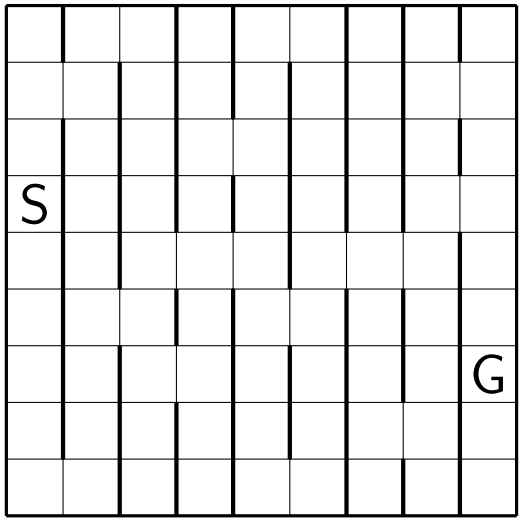
- One step extension is slightly more complex:

```
– one_step_extensions( [Node|Path], NewPaths ) :-
    state_of( Node, State ),
    findall( [NewNode,Node|Path],
      ( state_change( Rule, State, NewState, HCost ),
        make_node( Rule, NewState, HCost, NewNode ) ),
      NewPaths ).
```

- `choose` and `add_to_paths` – like uniform-cost search

BF: Operation

- What are possible heuristics?
  - Straight line distance
  - Manhattan distance



BF: Evaluation

- Completeness
  - No: infinite paths with $h(n) = h(successor(n))$
  - Oscillations (and no loop-checking)
- Optimality
  - No: like depth-first: tends to 'fixate' on a single path to the goal
- Complexity
  - Time complexity: $\mathcal{O}(b^m)$
  - Space complexity: $\mathcal{O}(b^m)$
  - This is worst case complexity: actual time and space complexity can be substantially reduced with a good heuristic

A* Search

- Uniform-cost search expands the node with the least actual path cost from the start state to its state; it is optimal and complete, but has poor complexity characteristics

- Best-first search expands the node with the least estimated path cost from its state to a goal state; it is neither optimal nor complete, but can have reasonable complexity characteristics (with a good heuristic)

- What happens if we sum the two evaluation functions?

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

  - $g(n)$ gives the actual path cost from start state S to state of node n
  - $h(n)$ gives the estimated path cost from state of node n to nearest goal state G
  - $f(n)$ gives the estimated cost of the cheapest solution path from S to G through n

A*: Heuristics

- The heuristic should be **admissible**
  - An admissible heuristic is an evaluation function that **never overestimates** the actual cost of a solution path through n
  - If h is admissible, then $f(n)$ never over-estimates the actual cost of the best solution path through n

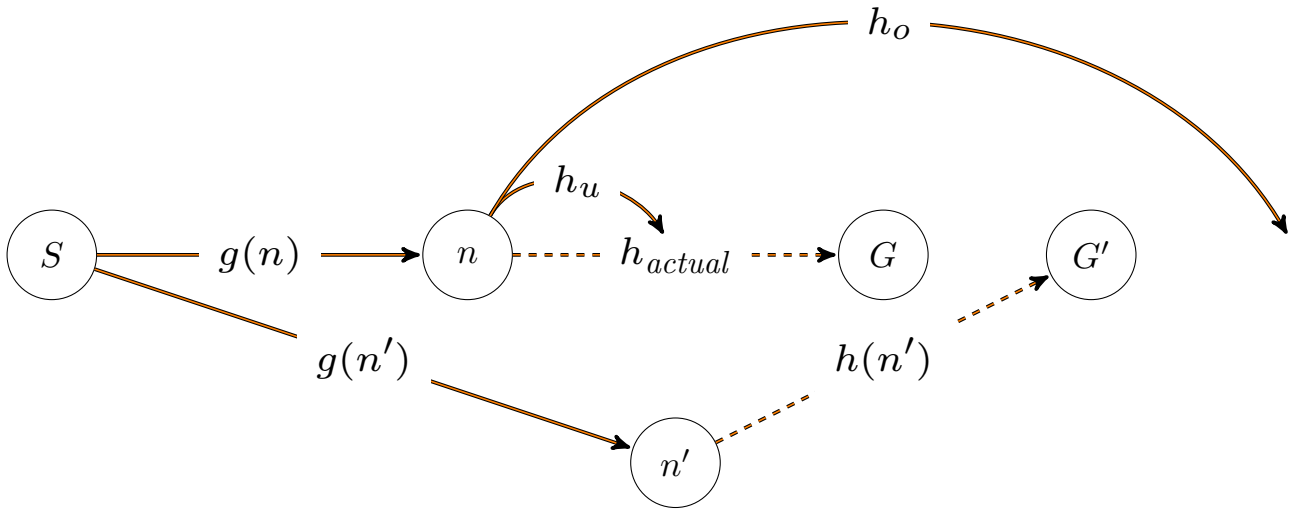
- Along any path, the f-cost never decreases
  - Most admissible heuristics are monotonic, and if not can be made monotonic by the **pathmax** equation

$$f(succ(n)) = max(f(n), g(succ(n)) + h(succ(n)))$$

- Therefore, A* expands the path whose frontier node has the least f-cost

A*: Never Over-Estimate

- Expanding the node with the lowest f-cost
  - If we estimate $h_o > h_{actual}$, so $g(n) + h_o > g(n') + h(n')$, then we might find a path from S to a sub-optimal goal G' through n'
  - If we estimate $h_u < h_{actual}$, at some point $g(n) + h_u < g(n') + h(n')$, so n will be expanded
  - If we always under-estimate, at some point we will expand the solution path ending in G before any solution path ending in G'



A* Algorithm

- Let f^* be the $actual$ cost of the optimal path from the start state to the goal state, so $f^* = f(G)$.

- Then:
  - A* will expand all nodes n for which $f(n) < f^*$
  - A* may expand some nodes n for which $f(n) = f^*$
  - A* will expand all nodes n for which $f(n) = f^*$, therefore including the goal node, before it expands any node for which $f(n) > f^*$

A*: Declarative Specification

- Node representation includes both state, actual cost and estimated cost
- State change rule implements cost function g and heuristic function h

```
state_change( Rule, State, NewState, GCost, HCost ) :-
  ..., g(..., GCost), h(..., HCost), ...
```

- One step extension has to add everything up:

```
one_step_extensions( [Node|Path], NewPaths ) :-
     state_of( Node, State ),
     gcost_of( Node, GPath ),
     findall( [NewNode,Node|Path],
       ( state_change( Rule, State, NewState, GCost, HCost ),
         Gactual is GPath + GCost,
         FCost is Gactual + HCost,
         make_node( Rule, NewState, Gactual, FCost, NewNode ) ),
       NewPaths ).
```

A*: Operation

- Possible heuristics
  - Straight line distance, or Manhattan distance
  - Why are these admissible heuristics?

- Every move, the g-cost goes up by 1. The h-cost never decreases by more than 1. Therefore, the f-cost along any path never decreases

- Intuitively
  - The g-cost draws 'contours' in the search space
  - The h-cost distorts those contours
    * Without crossing them over
    * Become closer together along 'favourable' paths

A*: Evaluation – Optimality

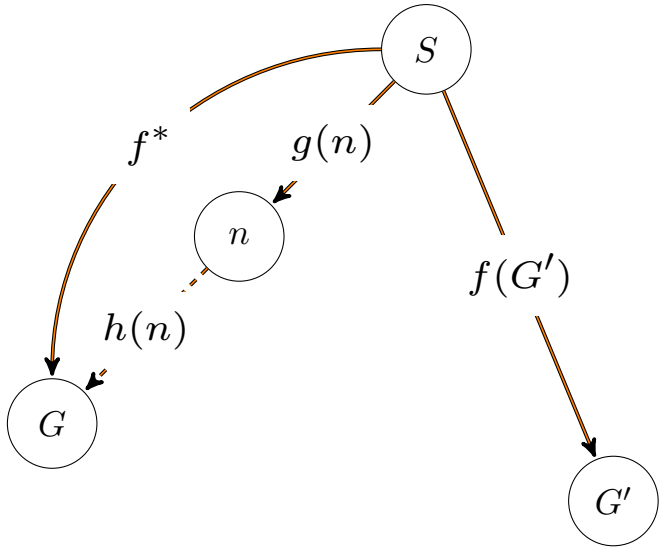
- Optimality
  - The first solution found by A* is provably optimal
  - A* is provably optimally efficient amongst search algorithms of this type

- Proof
  - Let G be an optimal goal state, with path cost f^*
  - Let G" be a sub-optimal goal state with path cost $f(G')$. Then

$$f(G') = g(g') + h(G') = g(G') + 0 = g(G')$$

  - Suppose A* returns G' as its (sub-optimal) solution, so $f(G') > f^*$, so $g(G') > f^*$.

– Now consider a node n on the frontier node on a path leading to G



$$f^* \geq f(n)$$ h is admissible and the path cost is monotonically increasing

$$f(n) \geq f(G')$$ otherwise n would have been selected for expansion before G'

$$f^* \geq f(G')$$ transitivity

$$f^* \geq g(G')$$

  - This is a contradiction
  - So either G' was optimal, or A* never expands a sub-optimal goal

- Optimally efficient
  - No other optimal algorithm is guaranteed to expand fewer nodes than A*
  - Any algorithm that does not expand every node with an f-cost less than f^* for a given h runs the risk of missing the optimal solution
  - For the proof, see Dechter and Pearl (1985)

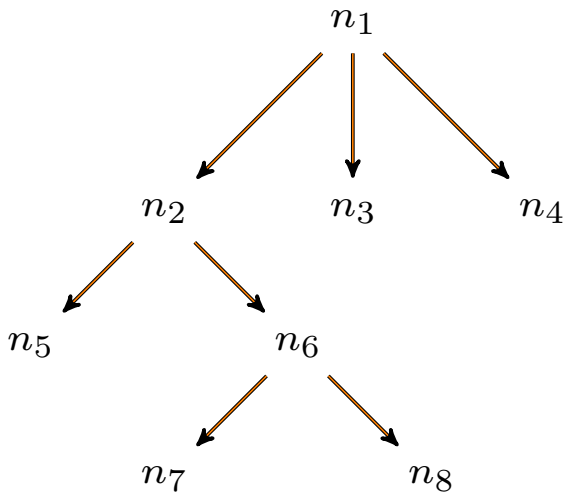
A*: Evaluation – Completeness and Complexity

- Completeness
  - A* expands nodes in order of increasing f-cost
  - Each expansion should have a lower bound > 0
  - So A* must eventually expand all nodes such that $f(n) \leq f^*$, at least one of which must be a goal state

- Complexity
  - For most problems, the number of nodes expanded is still exponential in the length of the path
  - Can get sub-exponential growth for certain types of heuristic/problem
    - "if the error in the heuristic function grows no faster than the logarithm of the actual path cost"
  - The main problem is still storing an exponentially increasing number of nodes

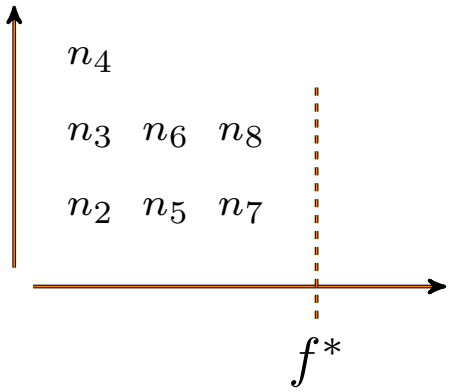
Aside: On Heuristics

- Consider: alternative candidates for the 8-puzzle
  - Heuristic h_1: count the number of tiles out of place
    - Why is this an admissible heuristic?
  - Heuristic h_2: count the Manhattan Distance for tiles out of place
    - Why is this an admissible heuristic?
  - Which heuristic is better, h_1 or h_2?

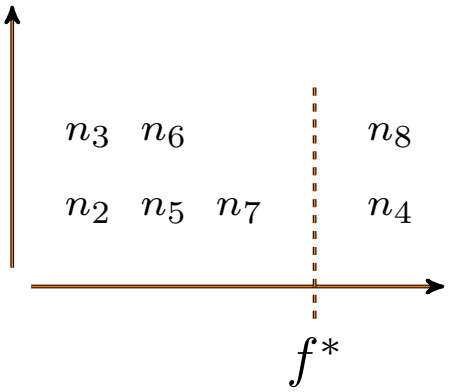


- Imagine drawing a histogram with increasing f-cost along the x-axis
  - Then map all the nodes onto the histogram
  - A* will expand all those nodes to the left of the f^* bar
  - Therefore the 'trick' is to get as many nodes to the right of the f^* bar ...
  - ... without ever over-estimating the h-cost

Suppose with h_1



But with h_2



Heuristic Accuracy & Efficiency

- Make $h(n)$ as large as possible (without over-estimating)
  - A* expands all nodes for which $f(n) < f^*$
  - So A* expands all nodes for which $h(n) < f^* - g(n)$
  - Therefore if $h_2(n) \geq h_1(n)$, for all n, h_2 may expand fewer nodes
- Minimize the effective branching factor b^*
- Combine heuristics
  - Without forgetting the complexity of computing the heuristics

Summary

- Breadth-first search was complete and optimal, with appalling complexity
- Depth-first search has better complexity, but not complete or optimal
- IDDF combined characteristics of both, but was still 'brute force'
- Uniform-cost search tried to take knowledge of the search space so far into account
- Beam search showed 'hacking' was not going to get anywhere
- Best-first tried to use knowledge of the expected search space
- A* combined uniform-cost and best-first, to give a provably optimal, and provably optimally efficient, search algorithm (of this kind)