CMSC 471
Artificial Intelligence

Search

KMA Solaiman – ksolaima@umbc.edu

Slide adoption courtesy: Tim Finin, Frank Ferraro (UMBC),
Charles R. Dyer (UW-Madison), Dan Klein and Pieter Abbeel (CS188, UC Berkeley)

A General Searching Algorithm

Core ideas:

1. Maintain a list of frontier (fringe) nodes
    1. Nodes coming *into* the frontier have been explored
    2. Nodes going out of the frontier have not been explored
2. Iteratively select nodes from the frontier and explore unexplored nodes from the frontier
3. Stop when you reach your goal

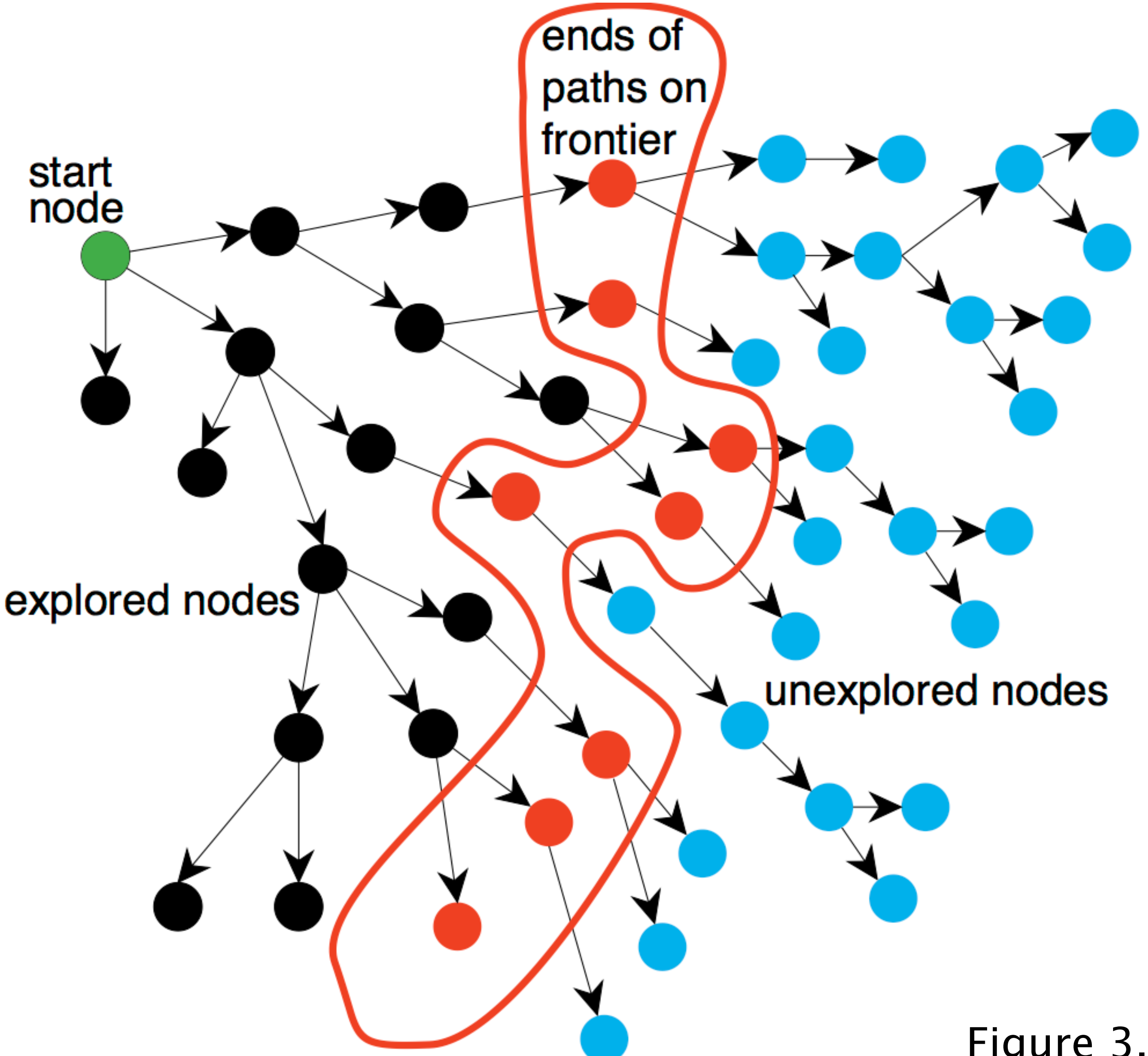


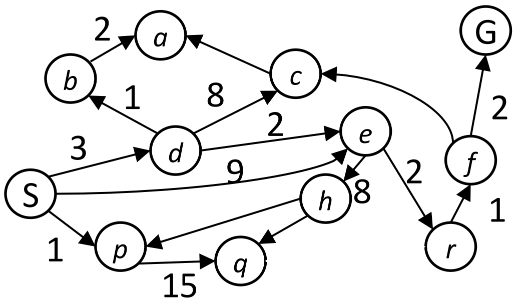
Figure 3.3

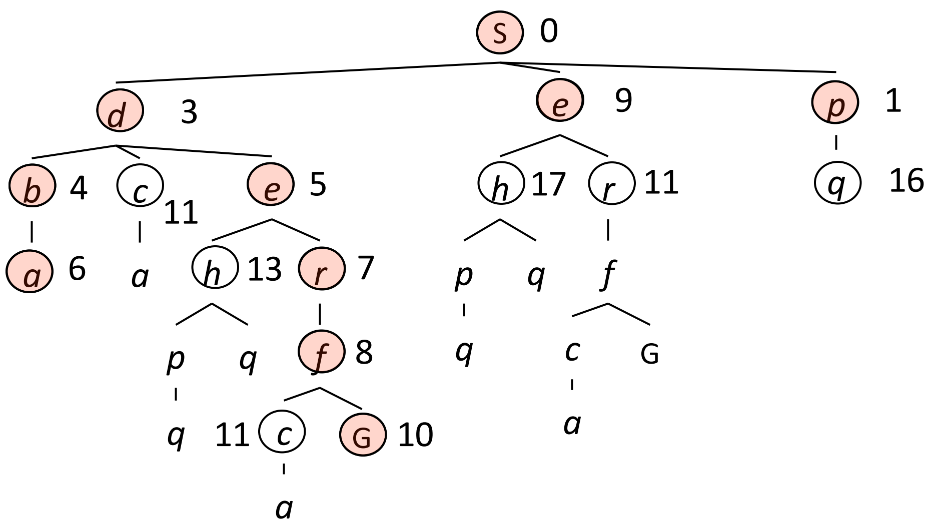
Uniform Cost Search

g(n) = cost from root to n

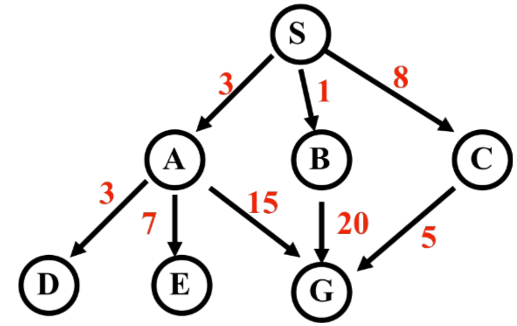
Strategy: expand lowest g(n)

Frontier is a priority queue sorted by g(n)





Uniform-Cost Search



Expanded node	Nodes list
	{ S^0 }
S^0	{ B^1 A^3 C^8 }
B^1	{ A^3 C^8 G^{21} }
A^3	{ D^6 C^8 E^{10} G^{18} G^{21} }
D^6	{ C^8 E^{10} G^{18} G^{21} }
C^8	{ E^{10} G^{13} G^{18} G^{21} }
E^{10}	{ G^{13} G^{18} G^{21} }
G^{13}	{ G^{18} G^{21} }

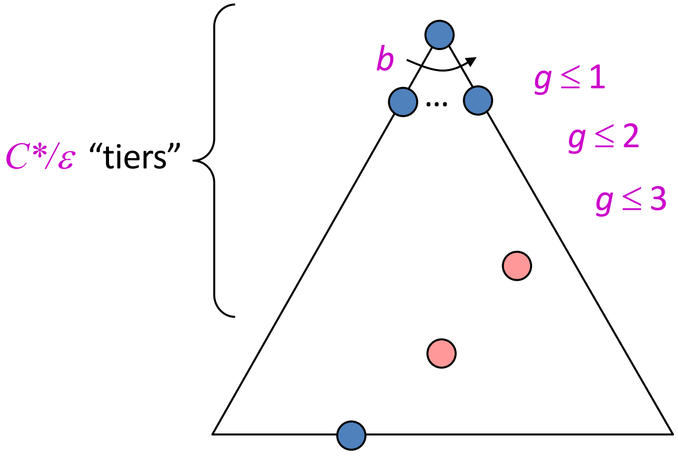
[priority queue](#)

Solution path found is S C G, cost 13

Number of nodes expanded (including goal node) = 7

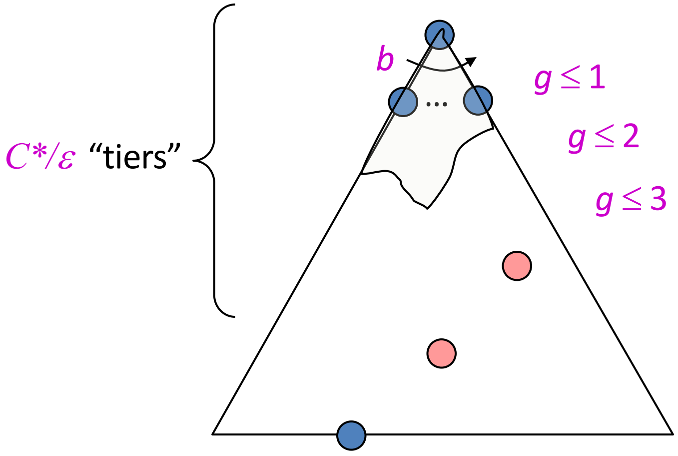
Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?



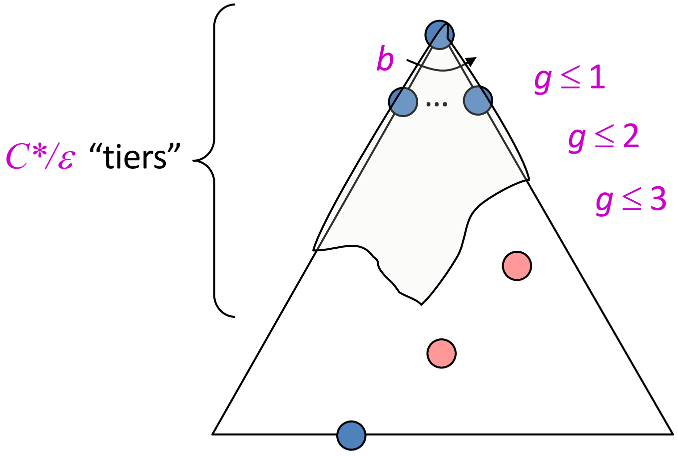
Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?



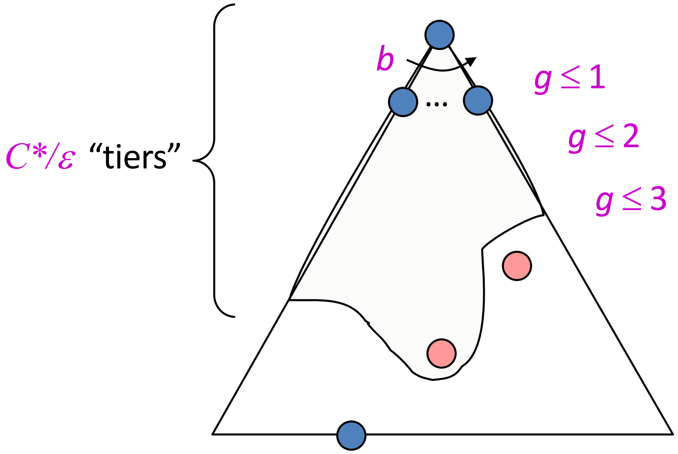
Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?



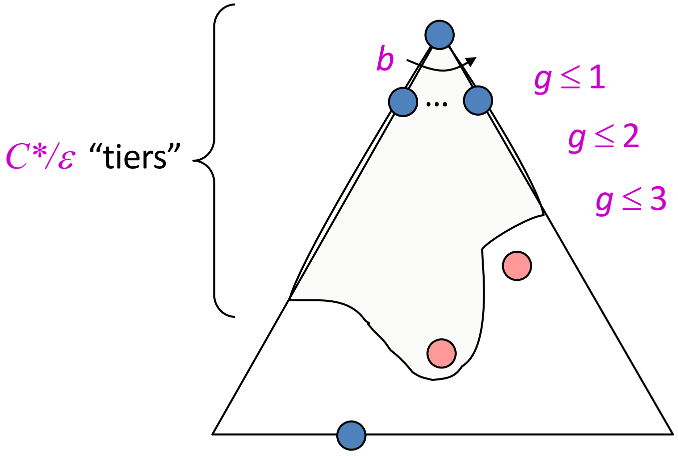
Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?



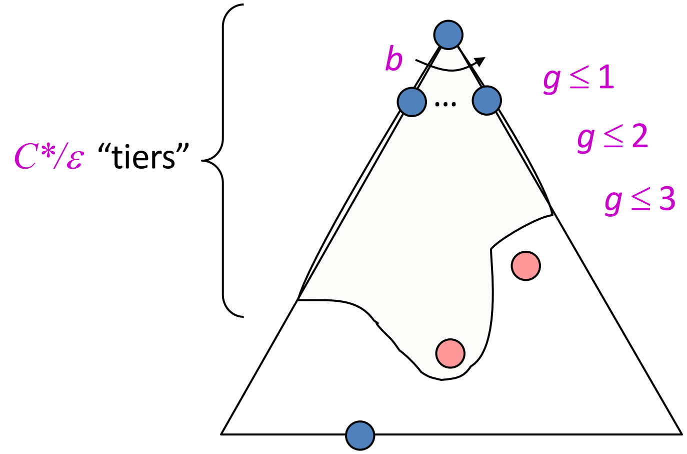
Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!



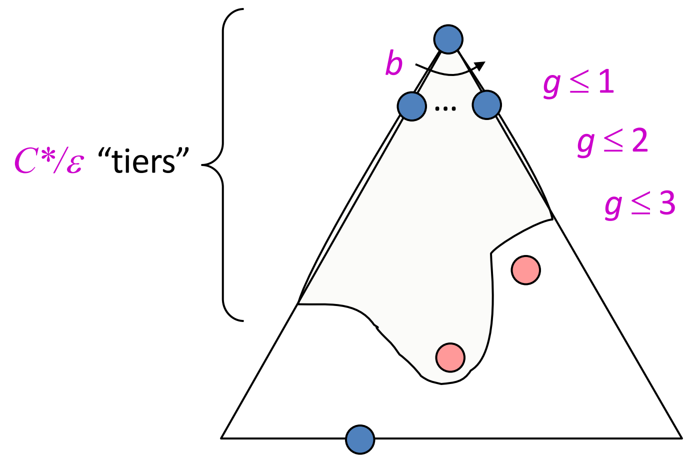
Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε



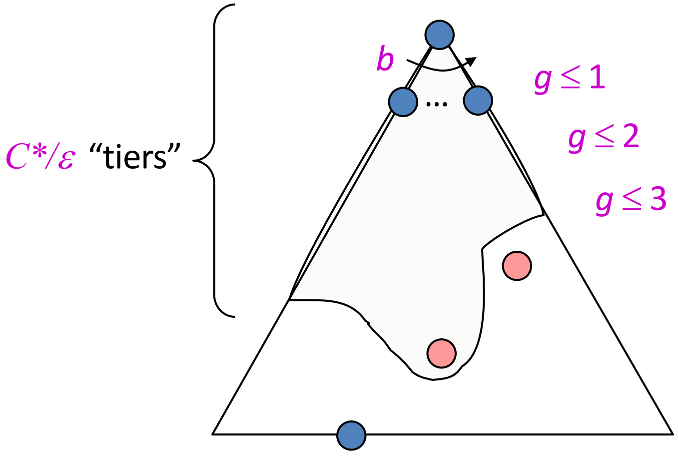
Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)



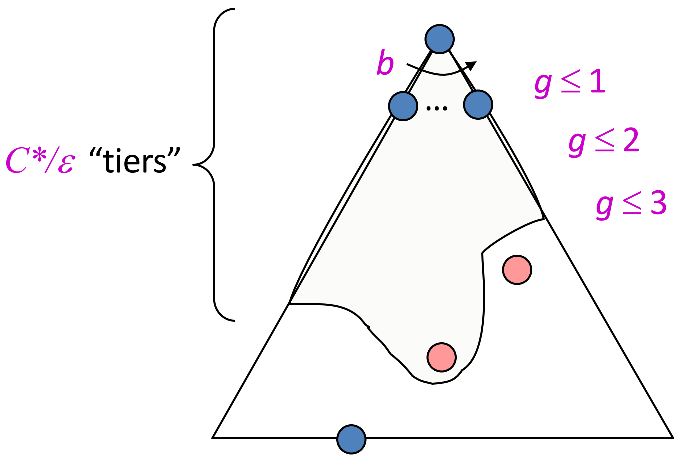
Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)
- How much space does the frontier take?



Uniform Cost Search (UCS) Properties

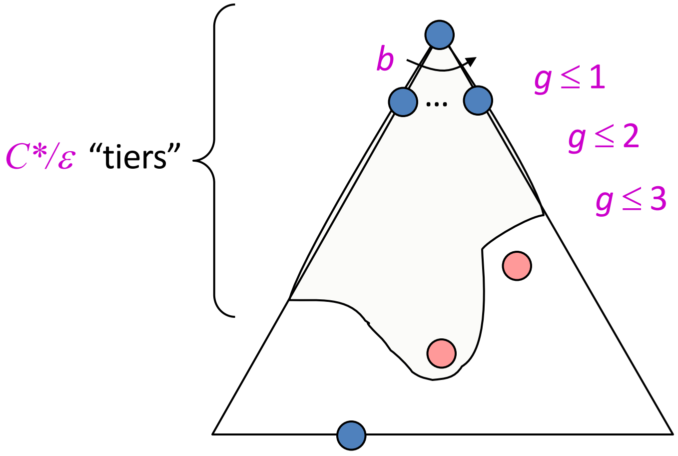
- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)



- How much space does the frontier take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$

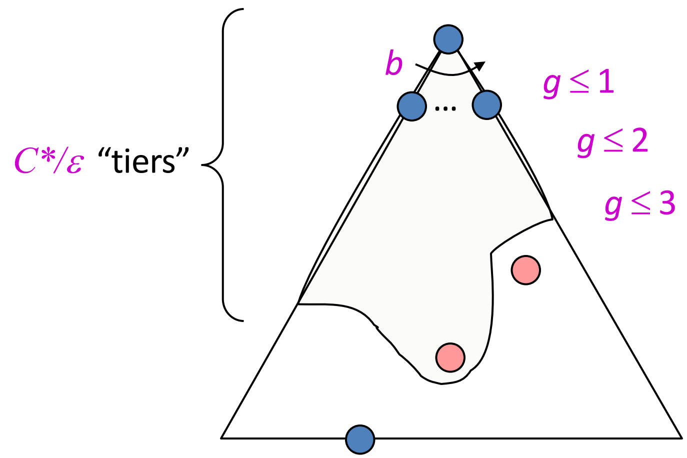
Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)
- How much space does the frontier take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$
- Is it complete?



Uniform Cost Search (UCS) Properties

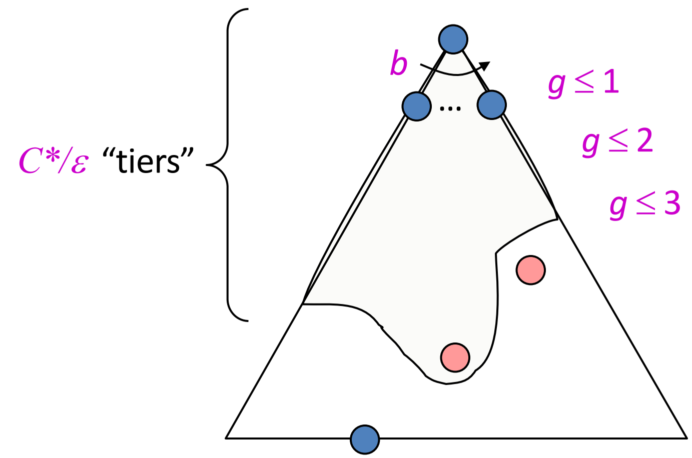
- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)



- How much space does the frontier take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$
- Is it complete?
  - Assuming C^* is finite and $\varepsilon > 0$, yes!

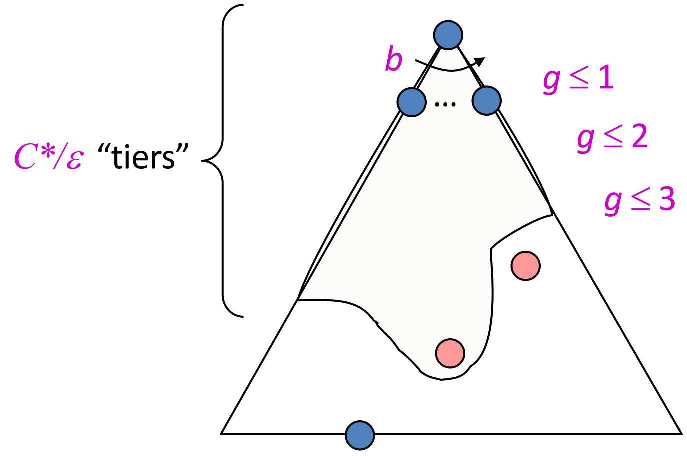
Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)
- How much space does the frontier take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$
- Is it complete?
  - Assuming C^* is finite and $\varepsilon > 0$, yes!
- Is it optimal?



Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)
- How much space does the frontier take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$
- Is it complete?
  - Assuming C^* is finite and $\varepsilon > 0$, yes!
- Is it optimal?
  - Yes! (Proof next lecture via A*)



Depth-First Iterative Deepening (DFID)

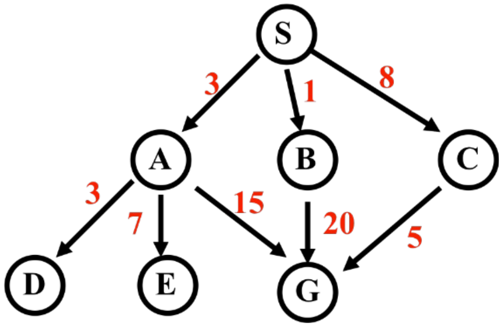
- Do DFS to depth 0, then (if no solution) DFS to depth 1, etc.
- Usually used with a tree search
- **Complete**
- **Optimal/Admissible** if all operators have unit cost, else finds shortest solution (like BFS)
- Time complexity a bit worse than BFS or DFS

  Nodes near top of search tree generated many times, but since almost all nodes are near tree bottom, worst case time complexity still exponential, $O(b^d)$

Depth-First Iterative Deepening (DFID)

- If branching factor is b and solution is at depth d, then nodes at depth d are generated once, nodes at depth d-1 are generated twice, etc.
  - Hence $b^d + 2b^{(d-1)} + \dots + db \leq b^d / (1 - 1/b)^2 = O(b^d)$.
  - If b=4, worst case is $1.78 * 4^d$, i.e., 78% more nodes searched than exist at depth d (in worst case)
- **Linear space complexity**, O(bd), like DFS
- Has advantages of BFS (completeness) and DFS (i.e., limited space, finds longer paths quickly)
- Preferred for **large state spaces** where **solution depth is unknown**

How they perform



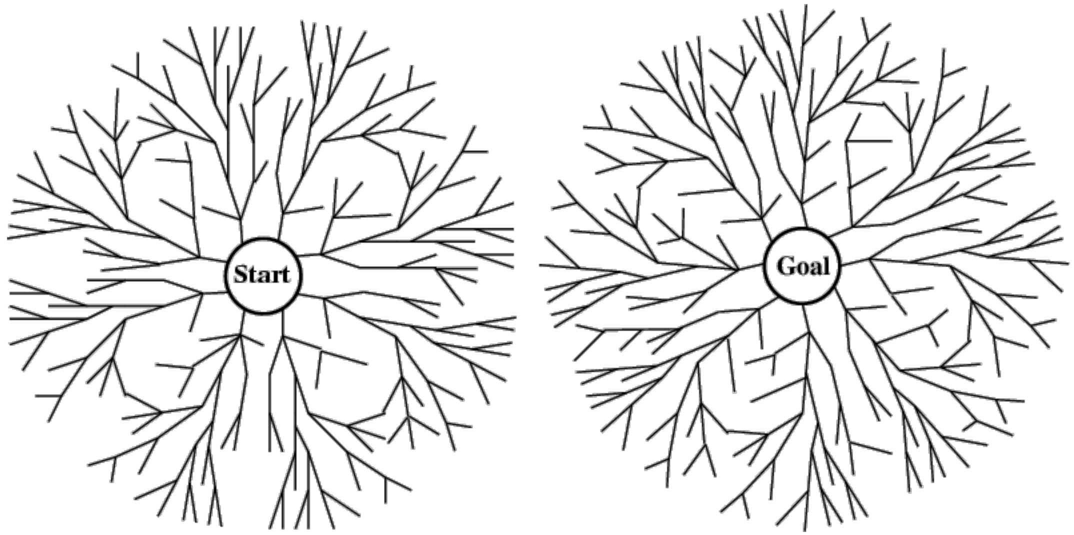
- **Depth-First Search:**
  - 4 Expanded nodes: S A D E G
  - Solution found: S A G (cost 18)
- **Breadth-First Search**:
  - 7 Expanded nodes: S A B C D E G
  - Solution found: S A G (cost 18)
- **Uniform-Cost Search**:
  - 7 Expanded nodes: S A D B C E G
  - Solution found: S C G (cost 13)

  Only uninformed search that worries about costs
- **Iterative-Deepening Search**:
  - 10 nodes expanded: S S A B C S A D E G
  - Solution found: S A G (cost 18)

Searching Backward from Goal

- Usually a successor function is reversible
  - i.e., can generate a node's predecessors in graph
- If we know a single goal (rather than a goal's properties), we could search backward to the initial state
- It might be more efficient
  - Depends on whether the graph fans in or out

Bi-directional search



- Alternate searching from the start state toward the goal and from the goal state toward the start
- Stop when the frontiers intersect
- Works well only when there are unique start & goal states
- Requires ability to generate “predecessor” states
- Can (sometimes) lead to finding a solution more quickly

Comparing Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

Summary

- Search in a problem space is at the heart of many AI systems
- Formalizing the search in terms of **states**, **actions**, and **goals** is key
- The simple “uninformed” algorithms we examined can be augmented to heuristics to improve them in various ways
- But for some problems, a simple algorithm is best

Informed (Heuristic) Search

- Heuristic search
- Best-first search
  - Greedy search
  - Beam search
  - A* Search
- Memory-conserving variations of A*
- Heuristic functions

Big idea: heuristic

Merriam-Webster's Online Dictionary

Heuristic (pron. \hy*u*-ˈris-tik\): adj. [from Greek *heuriskein* to discover] involving or serving as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods

The Free On-line Dictionary of Computing (15Feb98)

heuristic 1. <programming> A **rule of thumb**, simplification or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood. Unlike algorithms, heuristics do not guarantee feasible solutions and are often used with no theoretical guarantee. 2. <algorithm> **approximation algorithm**.

From WordNet (r) 1.6

heuristic adj 1: (CS) relating to or using a heuristic rule 2: of or relating to a general formulation that serves to guide investigation [ant: algorithmic] n : a **commonsense rule** (or set of rules) intended to increase the probability of solving some problem [syn: heuristic rule, heuristic program]

Heuristics, More Formally

$h(n)$ is a **heuristic function**, that maps a state n to an estimated cost from n-to-goal

Heuristics, More Formally

$h(n)$ is a **heuristic function**, that maps a state n to an estimated cost from n-to-goal

$h(n)$ is **admissible** iff $h(n) \leq$ the lowest actual cost from n-to-goal

Heuristics, More Formally

$h(n)$ is a **heuristic function**, that maps a state n to an estimated cost from n-to-goal

$h(n)$ is **admissible** iff $h(n) \leq$ the lowest actual cost from n-to-goal

$h(n)$ is **consistent** iff
$$h(n) \leq \text{lowestcost}(n, n') + h(n')$$

Informed methods add domain-specific information

- Select best path along which to continue searching
- h(n): estimates *goodness* of node n
- h(n) = **estimated cost** (or distance) of minimal cost path from n **to a goal state**.
- Based on domain-specific information and computable from current state description that estimates how close we are to a goal

Heuristics

- **All domain knowledge** used in search is encoded in the **heuristic function, h(<node>)**
- Examples:
  - 8-puzzle: number of tiles out of place
  - 8-puzzle: sum of distances each tile is from its goal
  - Missionaries & Cannibals: # people on starting river bank
- In general
  - $h(n) \geq 0$ for all nodes n
  - $h(n)$ = 0 implies that n is a goal node
  - $h(n) = \infty$ implies n is a dead-end that can't lead to goal

Heuristics for 8-puzzle

Misplaced Tiles Heuristic

(not including the blank)

Current State

3	2	8
4	5	6
7	1	

Goal State

1	2	3
4	5	6
7	8	

3	2	8
4	5	6
7	1	

3 tiles are not where they need to be

- Three tiles are misplaced (the 3, 8, and 1) so heuristic function evaluates to 3
- Heuristic says that it *thinks* a solution may be available in **3 or more** moves
- Very rough estimate, but easy to calculate

h = 3

Heuristics for 8-puzzle

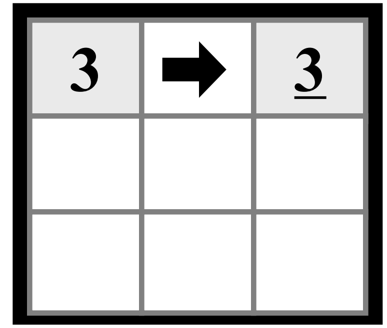
Manhattan Distance (not including the blank)

Current State

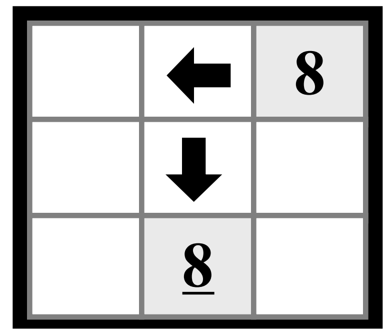
3	**2**	**8**
4	**5**	**6**
7	**1**	

Goal State

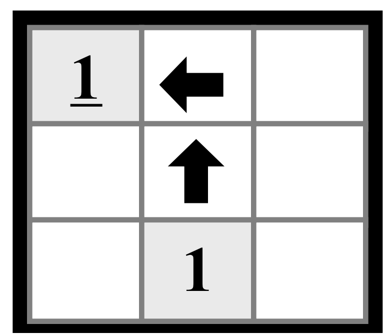
1	**2**	**3**
4	**5**	**6**
7	**8**	



2 spaces



3 spaces



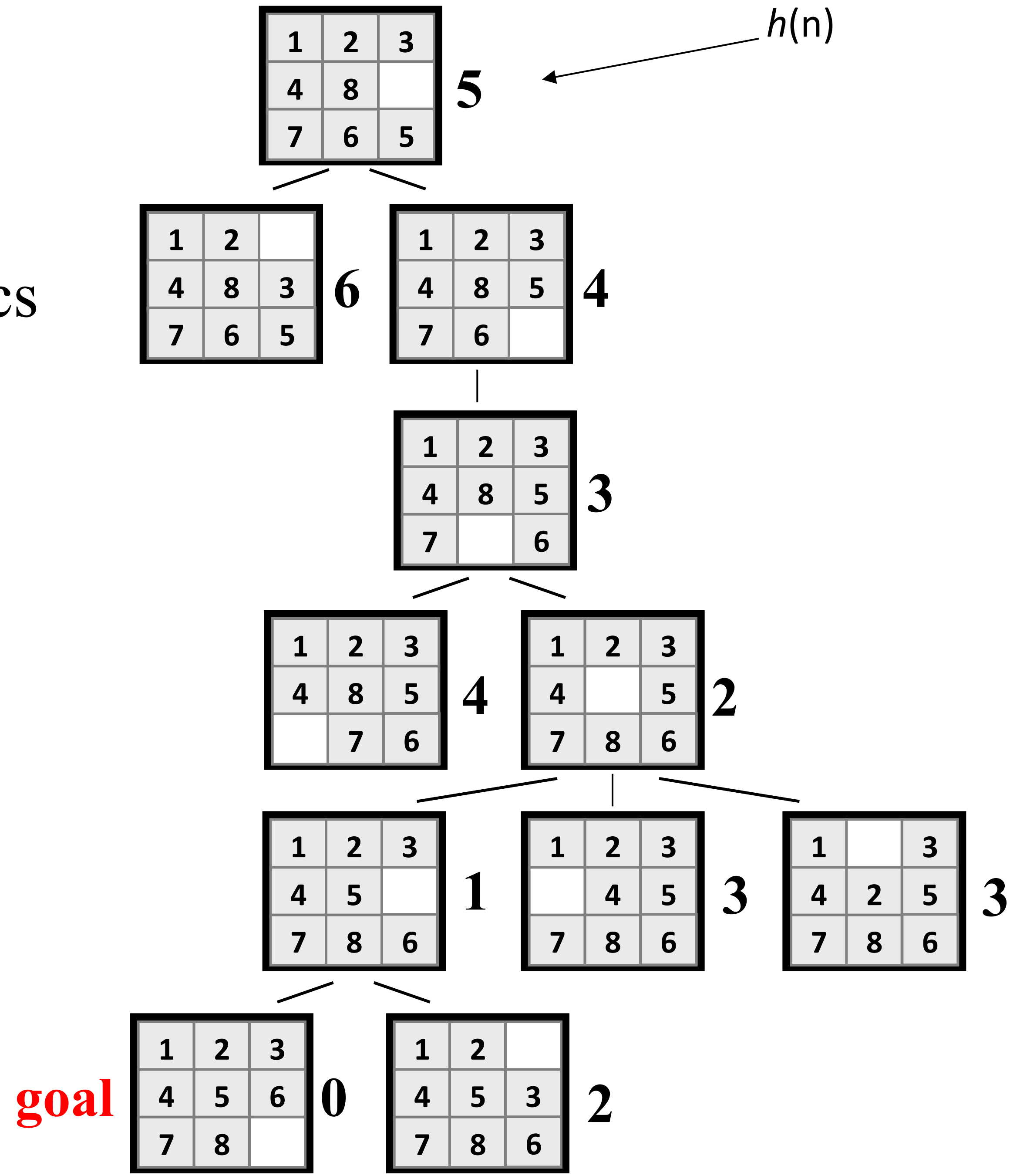
3 spaces

- The **3**, **8** and **1** tiles are misplaced (by 2, 3, and 3 steps) so the heuristic function evaluates to 8
- Heuristic says that it *thinks* a solution may be available in just 8 more moves.
- The misplaced heuristic's value is 3

Total 8

We can use heuristics to guide search

Manhattan Distance heuristic helps us quickly find a solution to the 8-puzzle



Best-first search

- Search algorithm that improves **depth-first search** by expanding most promising node chosen according to heuristic rule
- Order nodes on nodes list by increasing value of an evaluation function, **f(n)**, incorporating domain-specific information

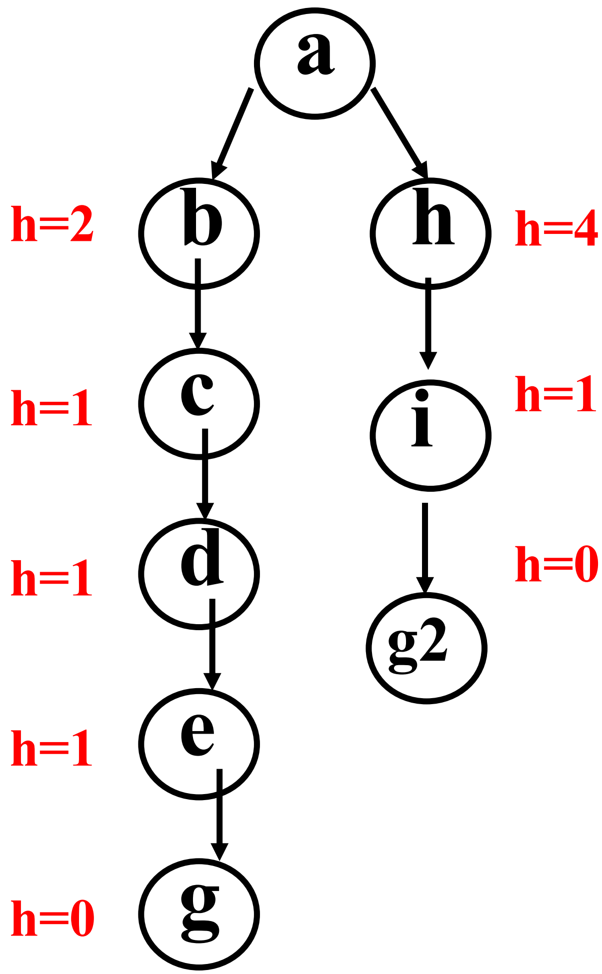
Best-first search

- Search algorithm that improves **depth-first search** by expanding most promising node chosen according to heuristic rule
- Order nodes on nodes list by increasing value of an evaluation function, **f(n)**, incorporating domain-specific information
- This is a generic way of referring to the class of informed methods

Greedy best first search

- A greedy algorithm makes locally optimal choices in hope of finding a global optimum
- Uses evaluation function $f(n) = h(n)$, sorting nodes by increasing values of f
- Selects node to expand appearing **closest** to goal (i.e., node with smallest f value)
- Not complete
- Not admissible, as in example
  - Assume arc costs = 1, greedy search finds goal g, with solution cost of 5
  - Optimal solution is path to goal with cost 3

Greedy best first search example



- Proof of non-admissibility
  - Assume arc costs = 1, greedy search finds goal g, with solution cost of 5
  - Optimal solution is path to goal with cost 3

Beam search

- Use evaluation function f(n), but maximum size of the nodes list is k, a fixed constant
- Only keep k best nodes as candidates for expansion, discard rest
- k is the *beam width*
- More space efficient than greedy search, but may discard nodes on a solution path
- As k increases, approaches best first search
- Complete?
- Admissible?

Beam search

- Use evaluation function f(n), but maximum size of the nodes list is k, a fixed constant
- Only keep k best nodes as candidates for expansion, discard rest
- k is the *beam width*
- More space efficient than greedy search, but may discard nodes on a solution path
- As k increases, approaches best first search
- Not complete
- Not admissible

We’ve *got* to be able to do better, right?

Let’s think about car trips…

A* Search

Use an evaluation function

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

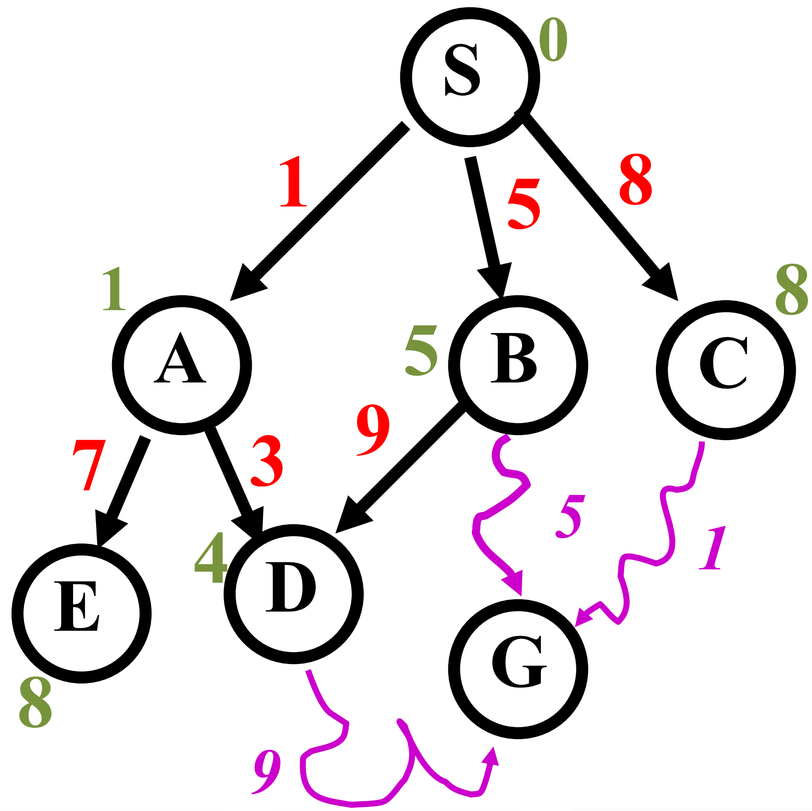
estimated total cost from start to goal via state n = minimal-cost path from the start state to state n + cost estimate from state n to the goal

A* Search

- Use as an evaluation function

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

- g(n) = minimal-cost path from the start state to state n
- g(n) adds "breadth-first" term to evaluation function
- Ranks nodes on search frontier by estimated cost of solution from start node ***via given node*** to goal



g(d)=4
h(d)=9
f(d)=13

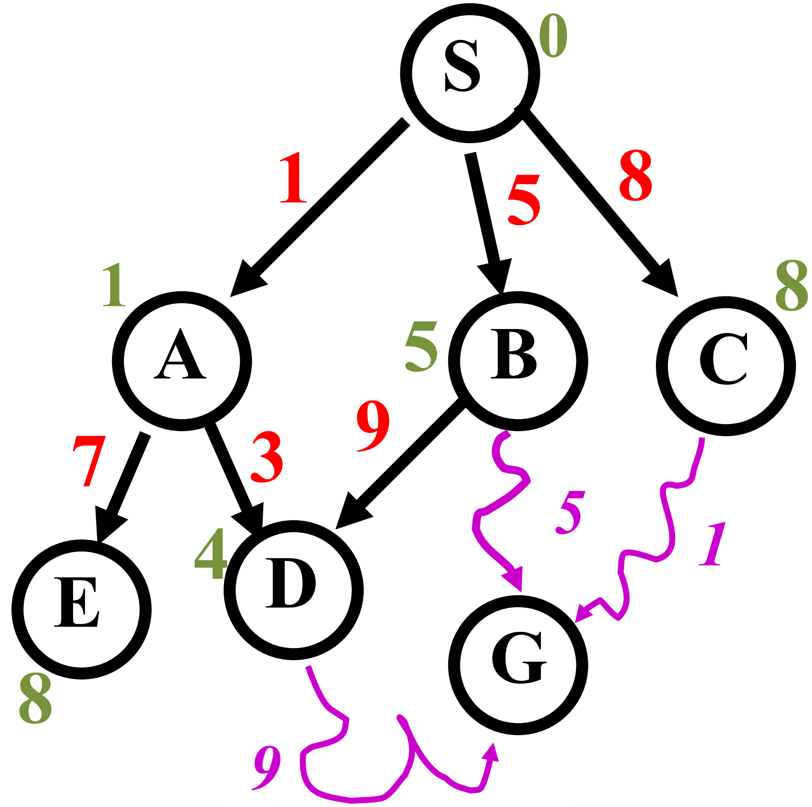
g(b)=5
h(b)=5
f(d)=10

g(c)=8
h(c)=1
f(c)=9

A* Search

- Use as an evaluation function

  f(n) = g(n) + h(n)
- g(n) = minimal-cost path from the start state to state n
- g(n) adds "breadth-first" term to evaluation function
- Ranks nodes on search frontier by estimated cost of solution from start node ***via given node*** to goal



g(d)=4	*g(b)=5*	*g(c)=8*
h(d)=9	*h(b)=5*	*h(c)=1*
f(d)=13	*f(d)=10*	*f(c)=9*

C is chosen next to expand

A* Search

- Use an evaluation function

$$\mathbf{f(n) = g(n) + h(n)}$$

estimated total cost from start to goal via state n = minimal-cost path from the start state to state n + cost estimate from state n to the goal

- g(n) term adds "breadth-first" component to evaluation function
- Ranks nodes on search frontier by estimated cost of solution from start node *via given node* to goal
- Not complete if h(n) can = ∞
- Is it admissible?

A*

- Pronounced *"a star"*
- h is **admissible** when h(n) <= h*(n) holds
  - **h*(n)** = *true cost* of *minimal cost path* from n to a goal
- Using an admissible heuristic guarantees that 1st solution found will be an **optimal** one
  - With an admissible heuristic, A* is cost-optimal
- A* is **complete** whenever branching factor is finite and every action has fixed, positive cost
- A* is **admissible**

Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *IEEE Transactions on Systems Science and Cybernetics SSC4* **4** (2): 100–107.

Implementing A*

Q: Can this be an instance of our general search algorithm?

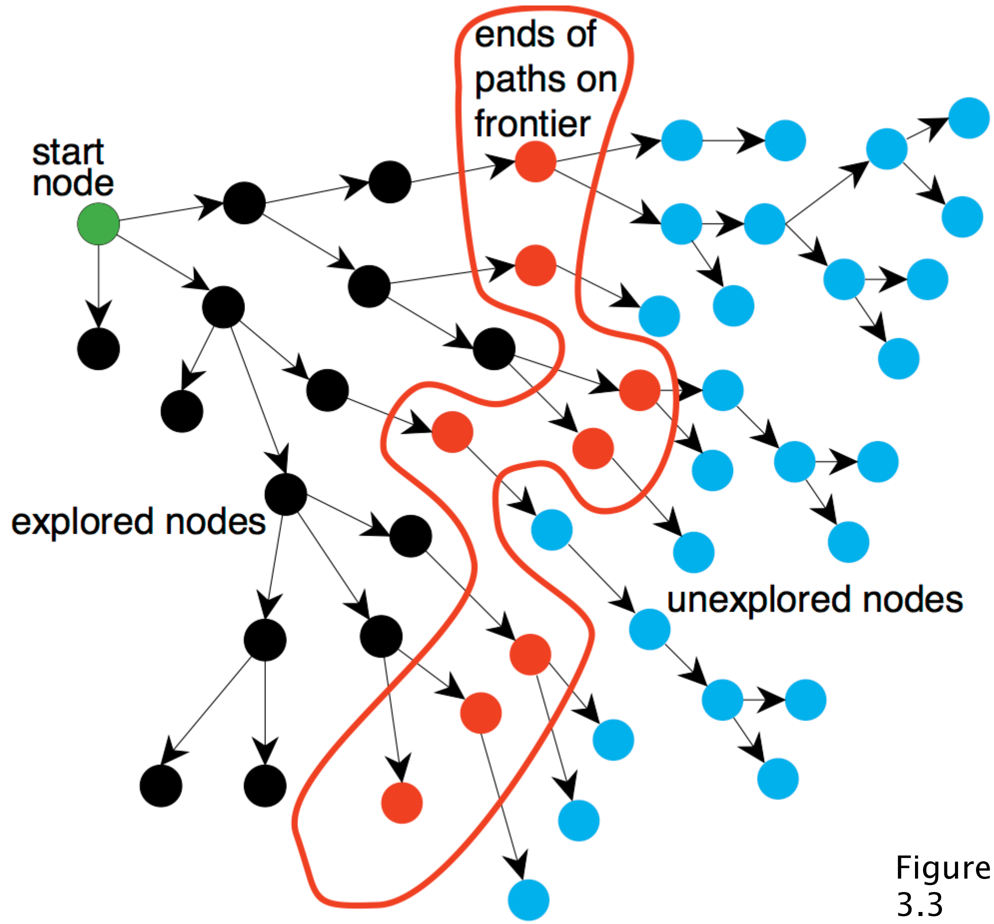


Figure 3.3

Implementing A*

Q: Can this be an instance of our general search algorithm?

A: Yup! Just make the fringe a priority queue ordered by $f(n)$

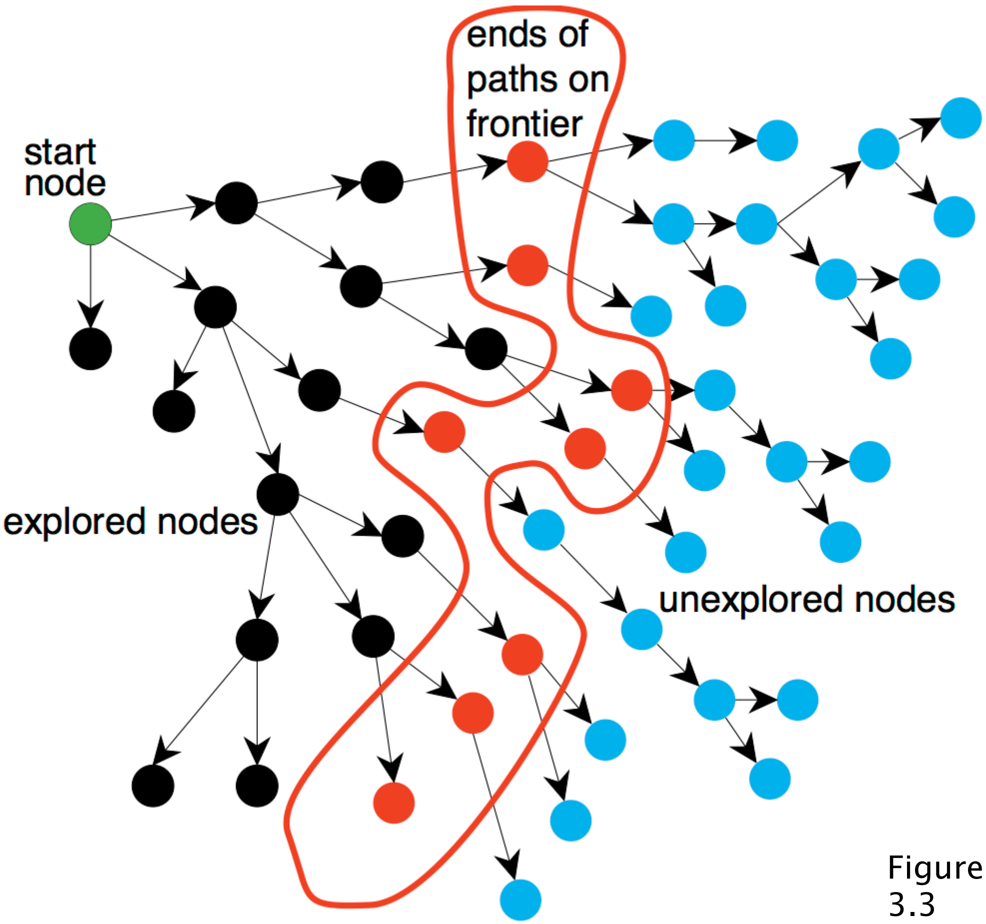


Figure 3.3

Alternative A* Pseudo-code

1 Put the start node S on the nodes list, called OPEN

2 If OPEN is empty, exit with failure

3 Select node in OPEN with minimal f(n) and place on CLOSED

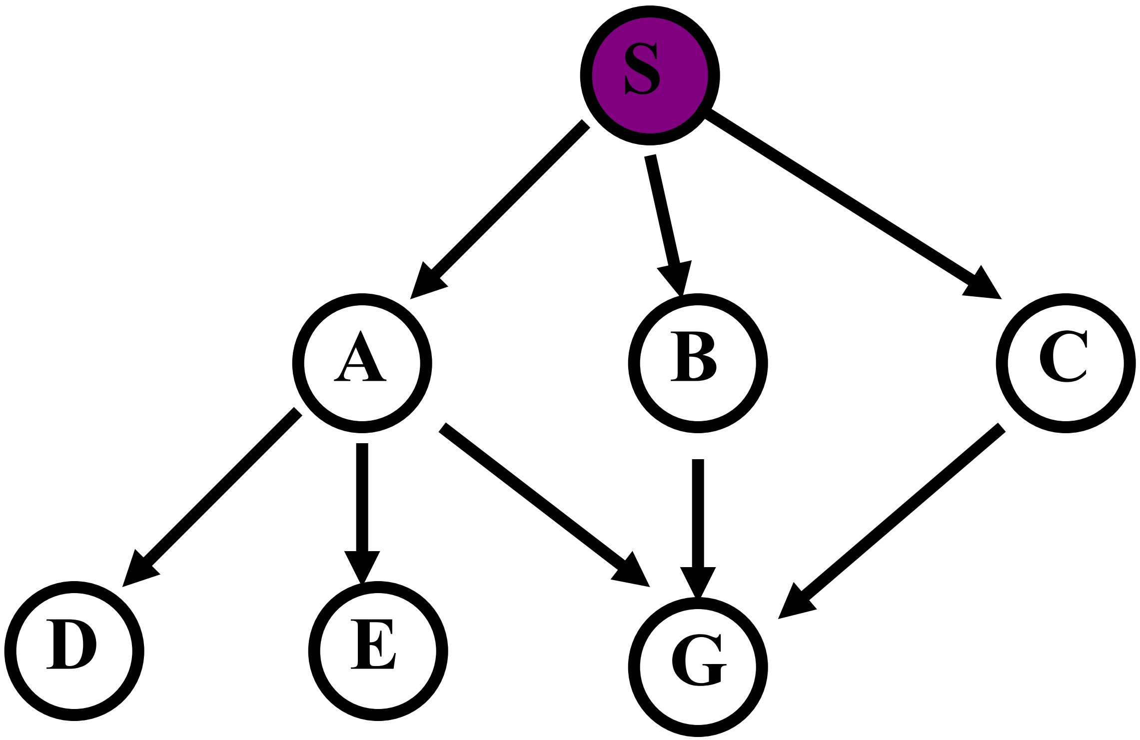
4 If n is a goal node, collect path back to start and stop

5 Expand n, generating all its successors and attach to them pointers back to n. For each successor n' of n

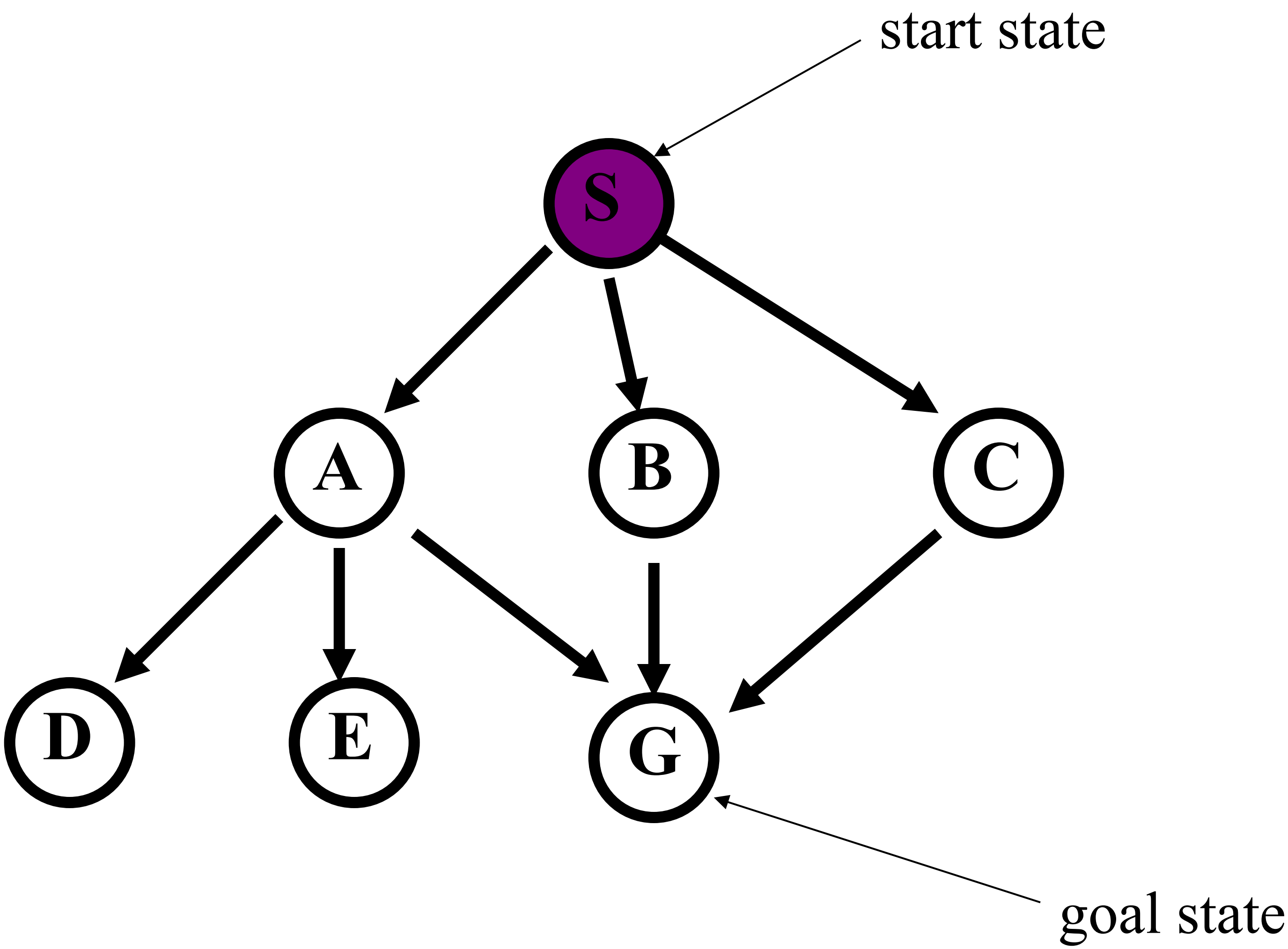
- **1** If n’ not already on OPEN or CLOSED
  - put n' on OPEN
  - compute h(n'), g(n')=g(n)+ c(n, n'), f(n')=g(n')+h(n')
- **2** If n’ already on OPEN or CLOSED and if g(n') is lower for new version of n', then:
  - Redirect pointers backward from n’ on path with lower g(n’)
  - Put n' on OPEN

IS A HEURISTIC ADMISSIBLE?

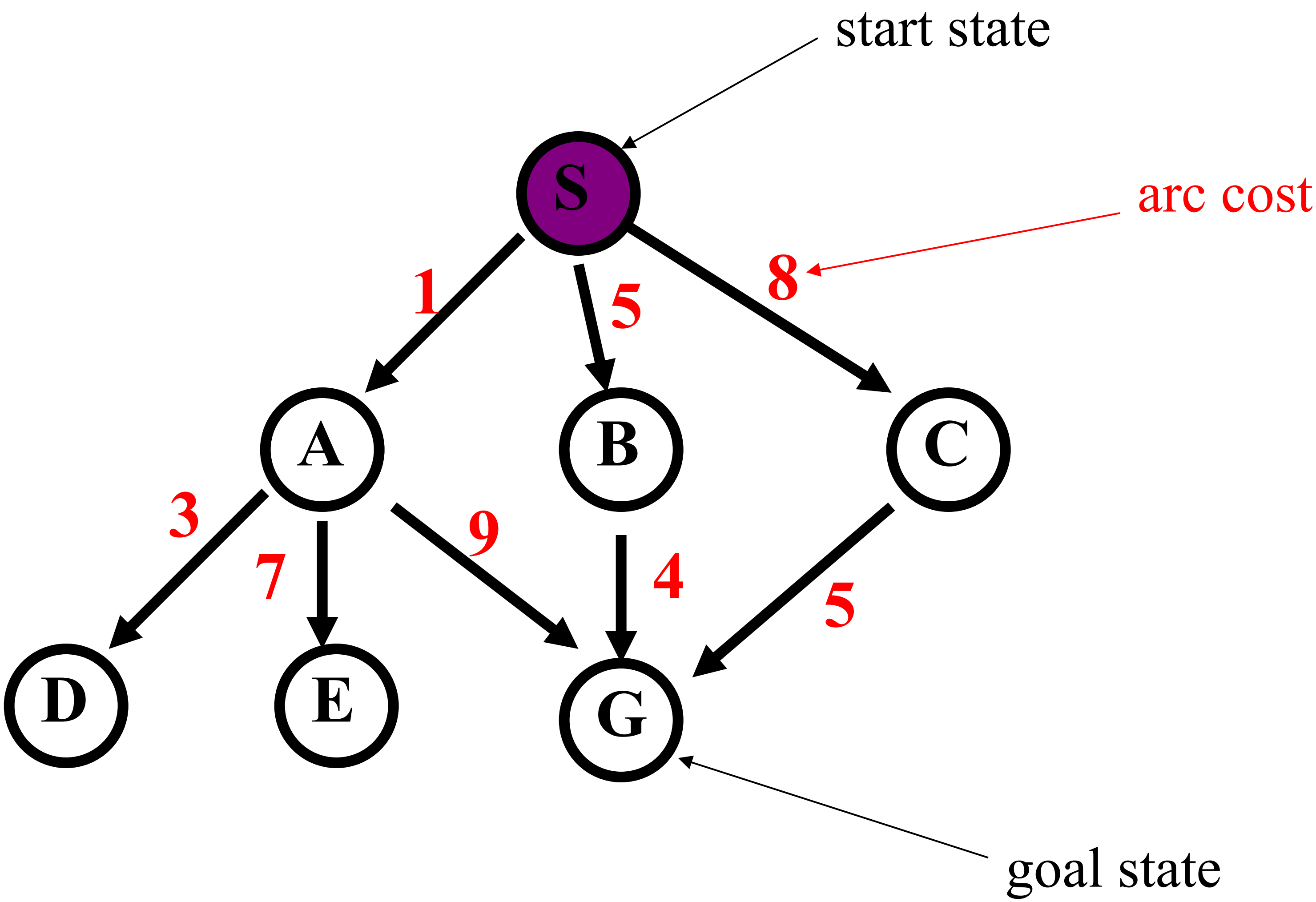
Example search space



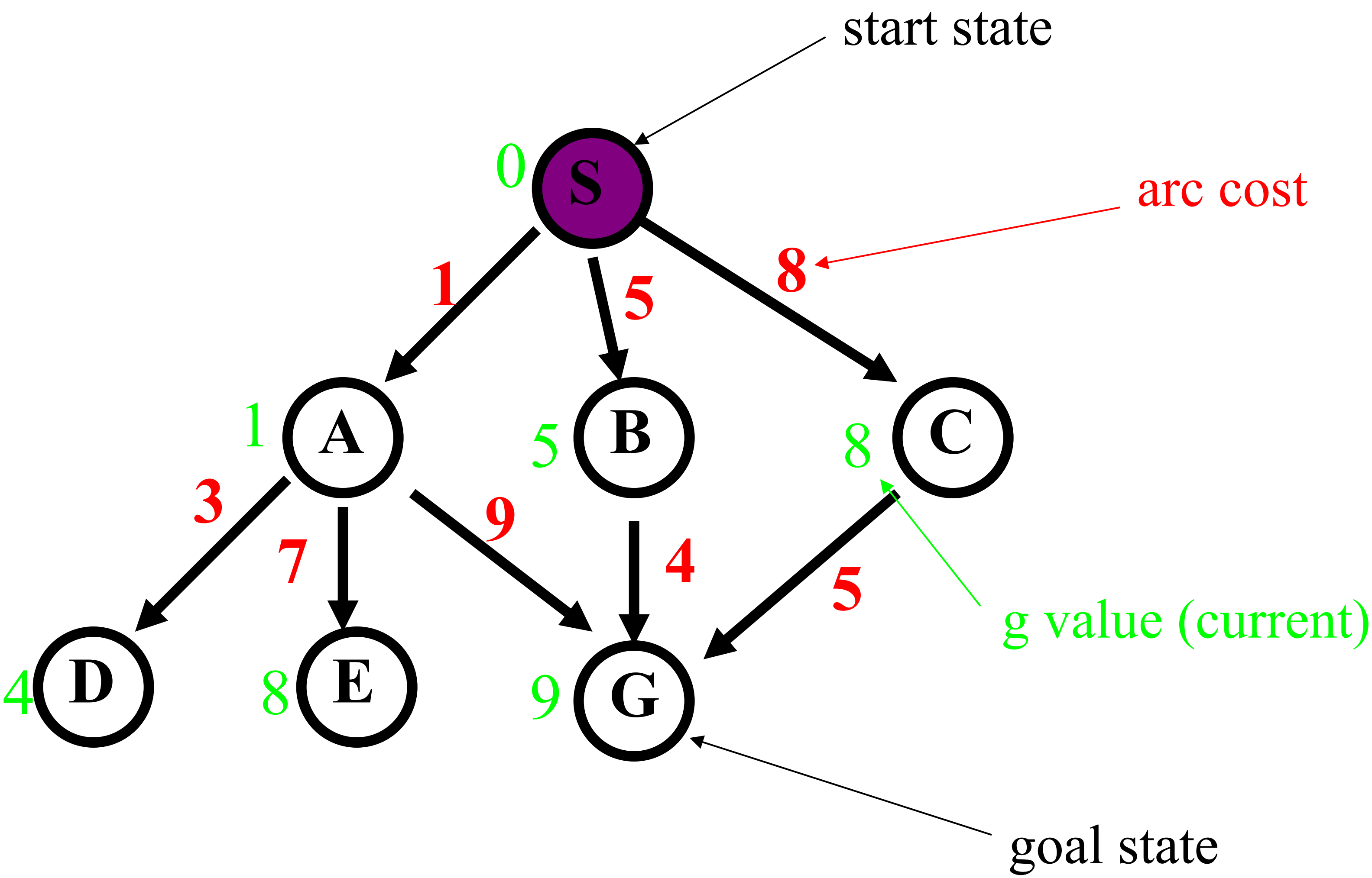
Example search space



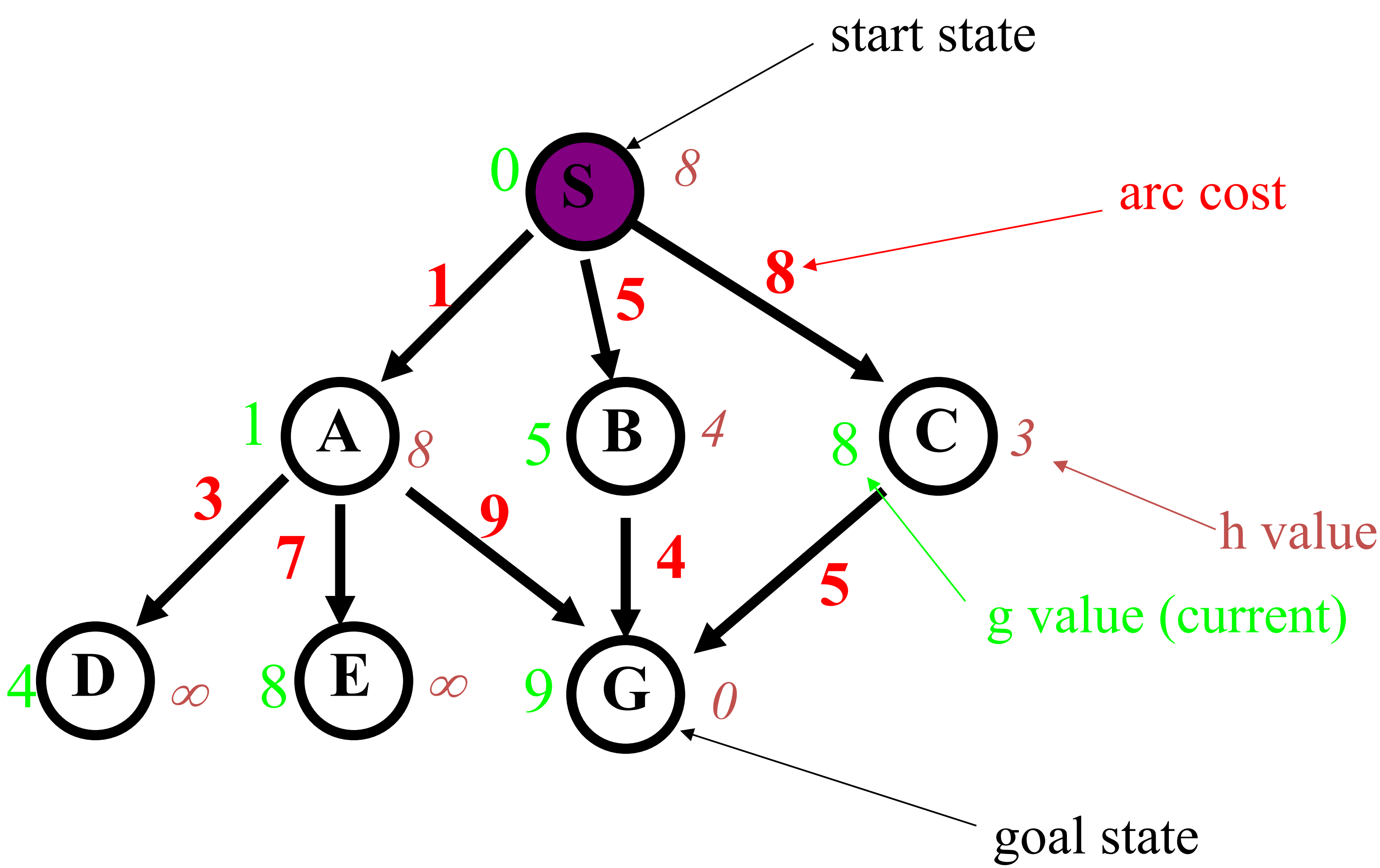
Example search space



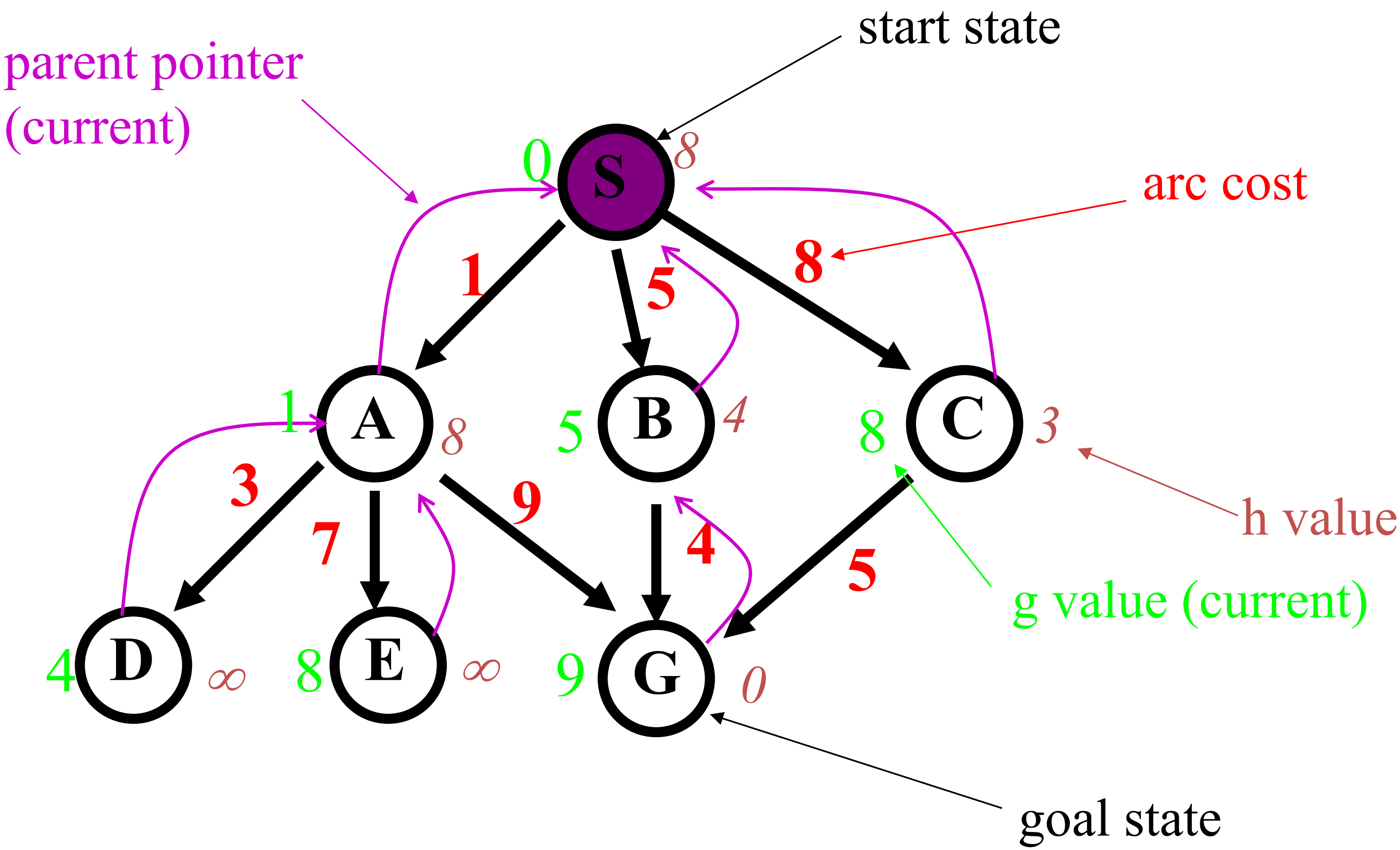
Example search space



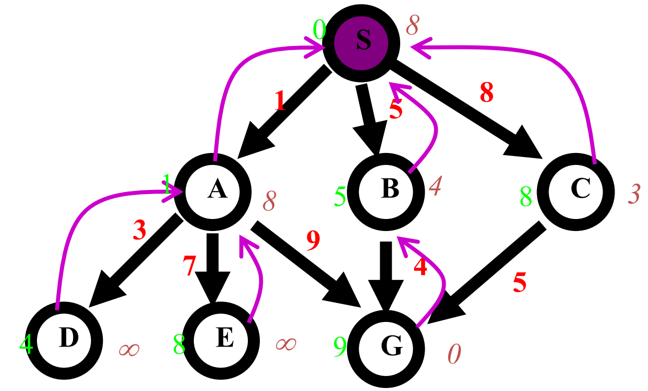
Example search space



Example search space



Example

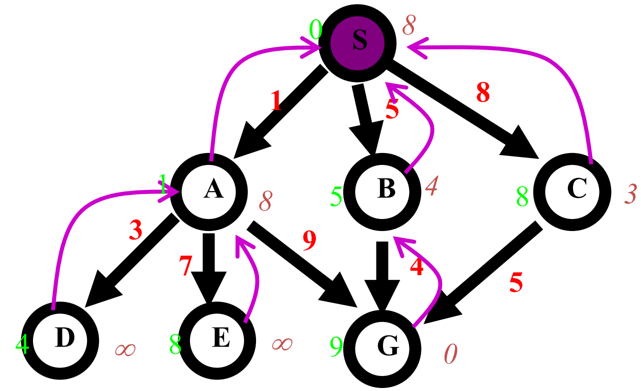


n	g(n)	h(n)	f(n)	h*(n)
S	0	8	8	9

- h*(n) is (hypothetical) perfect heuristic (an oracle)
- Since h(n) <= h*(n) for all n, h is admissible (optimal)
- Optimal path = *S B G* with cost 9

The table and graph show values for the entire space, but we must discover or compute them during the search

Example

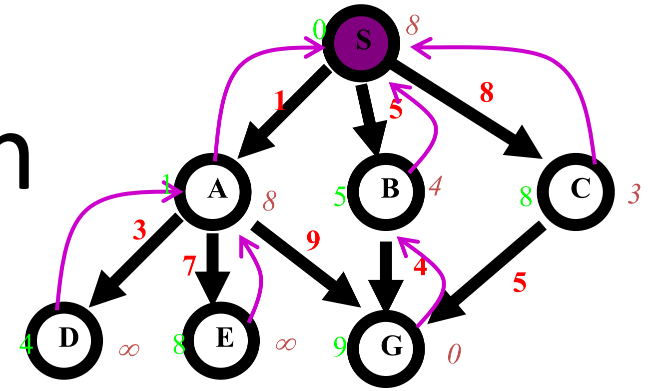


n	g(n)	h(n)	f(n)	h*(n)
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	inf	inf	inf
E	8	inf	inf	inf
G	9	0	9	0

- h*(n) is (hypothetical) perfect heuristic (an oracle)
- Since h(n) <= h*(n) for all n, h is admissible (optimal)
- Optimal path = *S B G* with cost 9

GREEDY VS A*

Greedy search



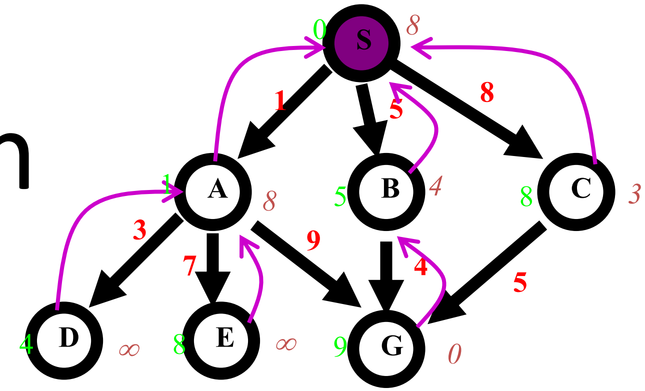
f(n) = h(n)

node expanded **nodes list**

{ S(8) }

what's next???

Greedy search

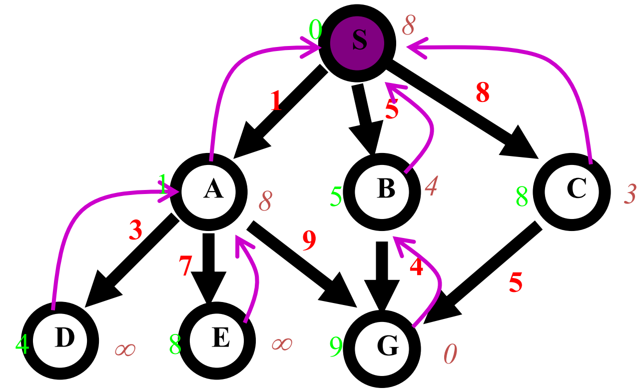


f(n) = h(n)

node expanded	**nodes list**
	{ S(8) }
S	{ C(3) B(4) A(8) }
C	{ G(0) B(4) A(8) }
G	{ B(4) A(8) }

- Solution path found is S C G, 3 nodes expanded.
- See how fast the search is!! But it is NOT optimal.

A* search



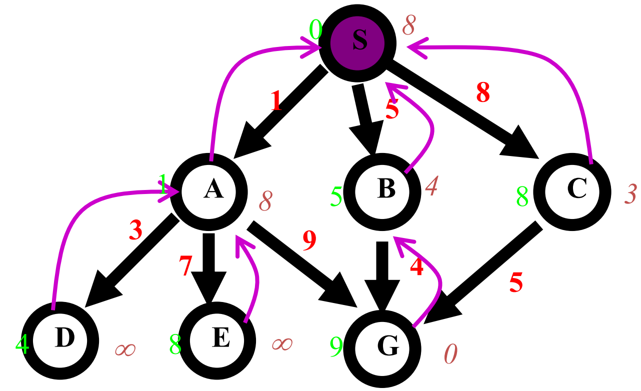
f(n) = g(n) + h(n)

node exp.   nodes list

{ S(8) }

What's next?

A* search



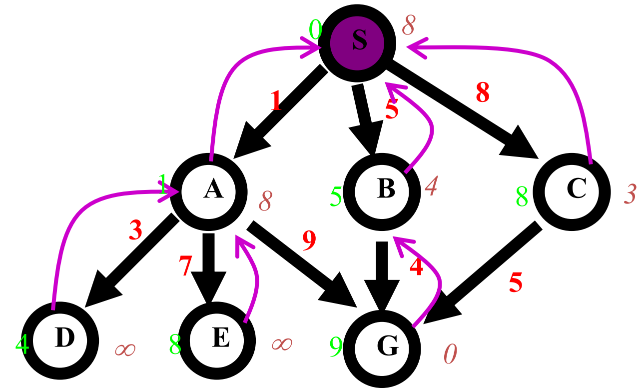
f(n) = g(n) + h(n)

```
node exp.       nodes list
             { S(8) }
 S           { A(9) B(9) C(11) }
                   What's next?
```

h(n)

h(S)=8
h(A)=8
h(B)=4
h(C)=3
h(D)=inf
h(E)=inf
h(G)=0

A* search

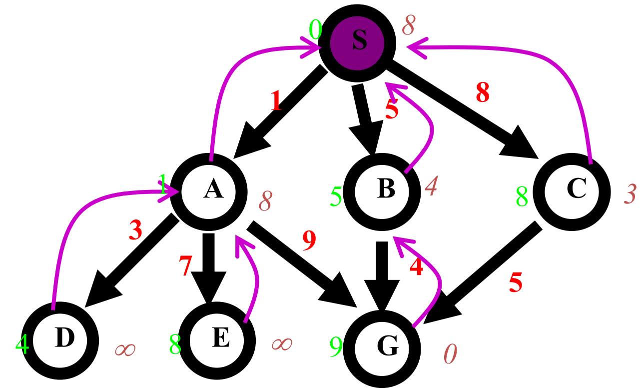


f(n) = g(n) + h(n)

node exp.	nodes list
	{ S(8) }
S	{ A(9) B(9) C(11) }
A	{ B(9) G(10) C(11) D(inf) E(inf) }

What's next?

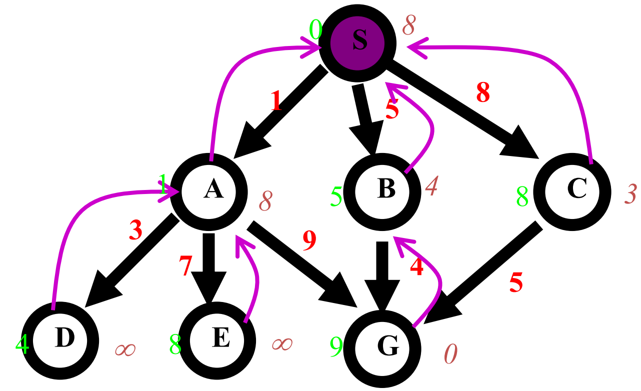
A* search



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

```
node exp.        nodes list
             { S(8) }
 S           { A(9) B(9) C(11) }
 A           { B(9) G(10) C(11) D(inf) E(inf) }
 B           { G(9) G(10) C(11) D(inf) E(inf) }
                   What's next?
```

A* search



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

```
node exp.      nodes list
               { S(8) }
 S             { A(9) B(9) C(11) }
 A             { B(9) G(10) C(11) D(inf) E(inf) }
 B             { G(9) G(10) C(11) D(inf) E(inf) }
 G             { C(11) D(inf) E(inf) }
```

- Solution path found is S B G, 4 nodes expanded..
- Still pretty fast. And optimal, too.

Observations on A*

- **Perfect heuristic:** If $h(n) = h^*(n)$ for all n, only nodes on an optimal solution path expanded; no extra work is done

Observations on A*

- **Perfect heuristic:** If h(n) = h*(n) for all n, only nodes on an optimal solution path expanded; no extra work is done
- **Null heuristic:** If h(n) = 0 for all n, then it is an admissible heuristic and A* acts like uniform-cost search

Observations on A*

- **Perfect heuristic:** If h(n) = h*(n) for all n, only nodes on an optimal solution path expanded; no extra work is done
- **Null heuristic:** If h(n) = 0 for all n, then it is an admissible heuristic and A* acts like uniform-cost search
- **Better heuristic:** If h1(n) < h2(n) <= h*(n) for all non-goal nodes, then h2 is a *better* heuristic than h1

Observations on A*

- **Perfect heuristic:** If h(n) = h*(n) for all n, only nodes on an optimal solution path expanded; no extra work is done
- **Null heuristic:** If h(n) = 0 for all n, then it is an admissible heuristic and A* acts like uniform-cost search
- **Better heuristic:** If h1(n) < h2(n) <= h*(n) for all non-goal nodes, then h2 is a *better* heuristic than h1
  - If A1* uses h1, and A2* uses h2, then every node expanded by A2* is also expanded by A1*

    i.e., A1 expands at least as many nodes as A2*
  - We say that A2* is *better informed* than A1*

Observations on A*

- **Perfect heuristic:** If h(n) = h*(n) for all n, only nodes on an optimal solution path expanded; no extra work is done
- **Null heuristic:** If h(n) = 0 for all n, then it is an admissible heuristic and A* acts like uniform-cost search
- **Better heuristic:** If h1(n) < h2(n) <= h*(n) for all non-goal nodes, then h2 is a *better* heuristic than h1
  - If A1* uses h1, and A2* uses h2, then every node expanded by A2* is also expanded by A1*

    i.e., A1 expands at least as many nodes as A2*
  - We say that A2* is *better informed* than A1*
- ***The closer h to h*, the fewer extra nodes expanded***

Summary (Fig 3.11)

Strategy	Selection from Frontier	Path found	Space
Breadth-first	First node added	Fewest arcs	Exponential
Depth-first	Last node added	No	Linear
Iterative deepening	—	Fewest arcs	Linear
Greedy best-first	Minimal $h(p)$	No	Exponential
Lowest-cost-first	Minimal cost (p)	Least cost	Exponential
A^*	Minimal cost $(p) + h(p)$	Least cost	Exponential
IDA^*	—	Least cost	Linear

Proof of the optimality of A*

- Assume that A* has selected G2, a goal state with a suboptimal solution, i.e., $g(G2) > f^*$
- Proof by contradiction shows it's impossible

Proof of the optimality of A*

- Assume that A* has selected G2, a goal state with a suboptimal solution, i.e., g(G2) > f*
- Proof by contradiction shows it's impossible
  - Choose a node n on an optimal path to G
  - Because h(n) is admissible, f* >= f(n)
  - If we choose G2 instead of n for expansion, then f(n) >= f(G2)
  - This implies f* >= f(G2)
  - G2 is a goal state: h(G2) = 0, f(G2) = g(G2).
  - Therefore f* >= g(G2)   => g(G2) <= f*
  - Contradiction

Dealing with hard problems

- For large problems, A* may require too much space
- Variations conserve memory: IDA* and SMA*
- IDA*, iterative deepening A*, uses successive iteration with growing limits on f, e.g.
  - A* but don't consider a node n where f(n) >10
  - A* but don't consider a node n where f(n) >20
  - A* but don't consider a node n where f(n) >30, ...
- SMA* -- Simplified Memory-Bounded A*
  - Uses queue of restricted size to limit memory use

IDA*: iterative deepening A*

Use successive iteration with growing limits on f, e.g.

- A* but don't consider a node n where f(n) >10
- A* but don't consider a node n where f(n) >20
- A* but don't consider a node n where f(n) >30, ...

SMA*: Simplified Memory-Bounded A*

Uses queue of restricted size to limit memory use

How to find good heuristics

Some options (mix-and-match):

- If h1(n) < h2(n) <= h*(n) for all n, h2 is better than h1
  - h2 **dominates** h1
- **Relaxing problem:** remove constraints for easier problem; use its solution cost as heuristic function
- Max of two admissible heuristics is a **Combining heuristics**: admissible heuristic, and it’s better!
- Use statistical estimates to compute h; may lose admissibility
- Identify good features, then use **machine learning** to find heuristic function; also may lose admissibility

Pruning: Dealing with Large Search Spaces

Cycle pruning

Multiple-path pruning

Don't add a node to the fringe if you've already expanded it (it's already on a path you've considered/are considering)

Q: What type of search-space would this be approach be applicable for?

Pruning: Dealing with Large Search Spaces

Cycle pruning

Don't add a node to the fringe if you've already expanded it (it's already on a path you've considered/are considering)

Q: What type of search-space would this be approach be applicable for?

Multiple-path pruning

Core idea: there may be multiple possible solutions, but you only need one

Maintain an "explored" (sometimes called "closed") set of nodes at the ends of paths; discard a path if a path node appears in this set

Q: Does this return an optimal solution?

Optimality with Multiple-Path Pruning

Some options to find the optimal solution (pulled from Ch 3.7.2)

- Make sure that the first path found to any node is a lowest-cost path to that node, then prune all subsequent paths found to that node. OR

Optimality with Multiple-Path Pruning

Some options to find the optimal solution (pulled from Ch 3.7.2)

- Make sure that the first path found to any node is a lowest-cost path to that node, then prune all subsequent paths found to that node. OR

- If the search algorithm finds a lower-cost path to a node than one already found, it could remove all paths that used the higher-cost path to the node. OR

Optimality with Multiple-Path Pruning

Some options to find the optimal solution (pulled from Ch 3.7.2)

- Make sure that the first path found to any node is a lowest-cost path to that node, then prune all subsequent paths found to that node. OR
- If the search algorithm finds a lower-cost path to a node than one already found, it could remove all paths that used the higher-cost path to the node. OR
- Whenever the search finds a lower-cost path to a node than a path to that node already found, it could incorporate a new initial section on the paths that have extended the initial path.

A* and Multiple-Path Pruning

If $h(n)$ is consistent, A* with multiple-path pruning will find an optimal solution

Core Idea: Why?

A* and Multiple-Path Pruning

If $h(n)$ is consistent, A* with multiple-path pruning will find an optimal solution

Core Idea: Why? (proof by contradiction: see Proposition 3.2 in Ch 3.7.2)

Summary: Informed search

- **Best-first search** is general search where minimum-cost nodes (w.r.t. some measure) are expanded first
- **Greedy search** uses minimal estimated cost h(n) to goal state as measure; reduces search time, but is neither complete nor optimal
- **A* search** combines uniform-cost search & greedy search: $f(n) = g(n) + h(n)$. Handles state repetitions & h(n) never overestimates
  - A* is complete & optimal, but space complexity high
  - Time complexity depends on quality of heuristic function
  - IDA* and SMA* reduce the memory requirements of A*

Summary (Fig 3.11)

Strategy	Selection from Frontier	Path found	Space
Breadth-first	First node added	Fewest arcs	Exponential
Depth-first	Last node added	No	Linear
Iterative deepening	—	Fewest arcs	Linear
Greedy best-first	Minimal $h(p)$	No	Exponential
Lowest-cost-first	Minimal $cost(p)$	Least cost	Exponential
A^*	Minimal $cost(p) + h(p)$	Least cost	Exponential
IDA^*	—	Least cost	Linear