16.410/413
Principles of Autonomy and Decision Making

Lecture 14: Informed Search

Emilio Frazzoli

Aeronautics and Astronautics
Massachusetts Institute of Technology

November 1, 2010

Outline

1. Informed search methods: Introduction
   - Shortest Path Problems on Graphs
   - Uniform-cost search
   - Greedy (Best-First) Search
2. Optimal search
3. Dynamic Programming

A step back

- We have seen how we can discretize collision-free trajectories into a finite graph.
- Searching for a collision-free path can be converted into a graph search.
- Hence, we can solve such problems using the graph search algorithms discussed in Lectures 2 and 3 (Breadth-First Search, Depth-First Search, etc.).

A step back

- We have seen how we can discretize collision-free trajectories into a finite graph.
- Searching for a collision-free path can be converted into a graph search.
- Hence, we can solve such problems using the graph search algorithms discussed in Lectures 2 and 3 (Breadth-First Search, Depth-First Search, etc.).

- However, roadmaps are not just “generic” graphs.
  - Some paths are much more preferable with respect to others *(e.g., shorter, faster, less costly in terms of fuel/tolls/fees, more stealthy, etc.).*
  - Distances have a physical meaning.
  - Good guesses for distances can be made, even without knowing optimal paths.

A step back

- We have seen how we can discretize collision-free trajectories into a finite graph.
- Searching for a collision-free path can be converted into a graph search.
- Hence, we can solve such problems using the graph search algorithms discussed in Lectures 2 and 3 (Breadth-First Search, Depth-First Search, etc.).

- However, roadmaps are not just "generic" graphs.
  - Some paths are much more preferable with respect to others *(e.g., shorter, faster, less costly in terms of fuel/tolls/fees, more stealthy, etc.)*.
  - Distances have a physical meaning.
  - Good guesses for distances can be made, even without knowing optimal paths.

Can we utilize this information to find efficient paths, efficiently?

Shortest Path Problems on Graphs

Input: $\langle V, E, w, \mathtt{start}, \mathtt{goal} \rangle$:

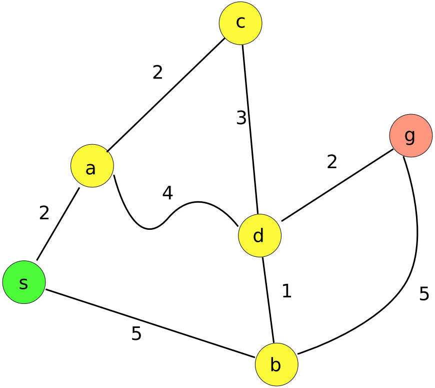
- V: (finite) set of vertices.
- $E \subseteq V \times V$: (finite) set of edges.
- $w : E \to \mathbb{R}_{>0}, e \mapsto w(e)$: a function that associates to each edge a strictly positive weight *(cost, length, time, fuel, prob. of detection)*.
- $\mathtt{start}, \mathtt{goal} \in V$: respectively, start and end vertices.

Output: $\langle P \rangle$

- P is a path (starting in `start` and ending in `goal`, such that its weight $w(P)$ is minimal among all such paths.
- The weight of a path is the sum of the weights of its edges.

Example: point-to-point shortest path

Find the minimum-weight path from s to g in the graph below:



Solution: a simple path $P = \langle g, d, a, s \rangle$ ($P = \langle g, d, b, s \rangle$ would be acceptable, too), with weight $w(P) = 8$.

Uniform-Cost Search

$Q \leftarrow \langle \texttt{start} \rangle$; `// Initialize the queue with the starting node`
while *Q is not empty* **do**
    Pick (and remove) the path P with lowest cost $g = w(P)$ from the queue Q ;
    if *head(P)* $=$ `goal` **then return** P ; `// Reached the goal`
    foreach *vertex v such that* $(head(P), v) \in E$, **do** //for all neighbors
        add $\langle v, P \rangle$ to the queue Q ; `// Add expanded paths`
return `FAILURE` ; `// Nothing left to consider.`

Uniform-Cost Search

$Q \leftarrow \langle \texttt{start} \rangle$; `// Initialize the queue with the starting node`

while *Q is not empty* **do**

  Pick (and remove) the path P with lowest cost $g = w(P)$ from the queue Q ;

  if $head(P) = \texttt{goal}$ **then return** P ; `// Reached the goal`

  foreach *vertex* v *such that* $(head(P), v) \in E$, **do** //for all neighbors

    add $\langle v, P \rangle$ to the queue Q ; `// Add expanded paths`

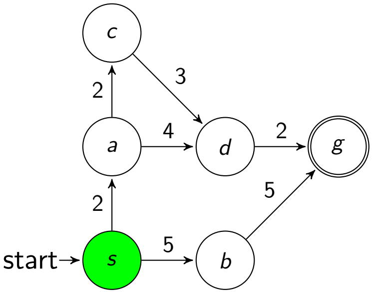
return `FAILURE` ; `// Nothing left to consider.`

Note: no visited list!

Example of Uniform-Cost Search: Step 1

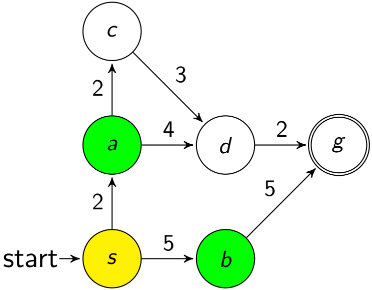
Q:

path	cost
$\langle s \rangle$	0



Example of Uniform-Cost Search: Step 2

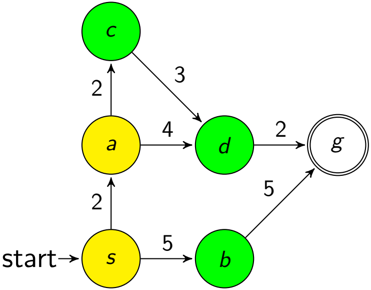
	path	cost
Q:	$\langle a, s \rangle$	2
	$\langle b, s \rangle$	5



Example of Uniform-Cost Search: Step 3

Q:

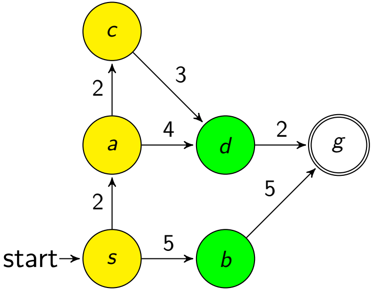
state	cost
$\langle c, a, s \rangle$	4
$\langle b, s \rangle$	5
$\langle d, a, s \rangle$	6



Example of Uniform-Cost Search: Step 4

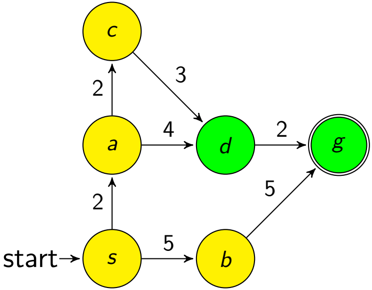
Q:

state	cost
$\langle b, s \rangle$	5
$\langle d, a, s \rangle$	6
$\langle d, c, a, s \rangle$	7



Example of Uniform-Cost Search: Step 5

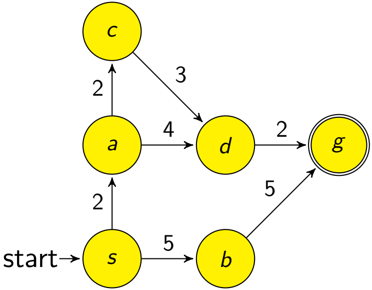
	state	cost
Q:	$\langle d, a, s \rangle$	6
	$\langle d, c, a, s \rangle$	7
	$\langle g, b, s \rangle$	10



Example of Uniform-Cost Search: Step 6

Q:

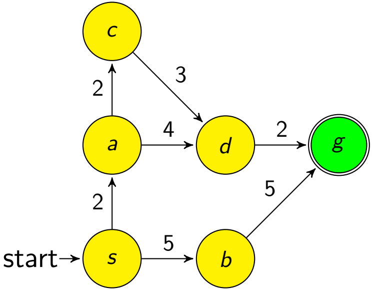
state	cost
$\langle d, c, a, s \rangle$	7
$\langle g, d, a, s \rangle$	8
$\langle g, b, s \rangle$	10



Example of Uniform-Cost Search: Step 7

Q:

state	cost
$\langle g, d, a, s \rangle$	8
$\langle g, d, c, a, s \rangle$	9
$\langle g, b, s \rangle$	10



Remarks on UCS

- UCS is an extension of BFS to the weighted-graph case (UCS = BFS if all edges have the same cost).
- UCS is complete and optimal (assuming costs bounded away from zero).
- UCS is guided by path cost rather than path depth, so it may get in trouble if some edge costs are very small.
- Worst-case time and space complexity $O\left(b^{W^*/\epsilon}\right)$, where W^* is the optimal cost, and ϵ is such that all edge weights are no smaller than ϵ.

Greedy (Best-First) Search

- UCS explores paths in all directions, with no bias towards the goal state.
- What if we try to get "closer" to the goal?
- We need a measure of distance to the goal. It would be ideal to use the length of the shortest path... but this is exactly what we are trying to compute!
- We can estimate the distance to the goal through a **"heuristic function,"** $h : V \to \mathbb{R}_{\geq 0}$. In motion planning, we can use, e.g., the Euclidean distance to the goal (as the crow flies).
- A reasonable strategy is to always try to move in such a way to minimize the estimated distance to the goal: this is the basic idea of the **greedy (best-first) search**.

Greedy (Best-First) Search

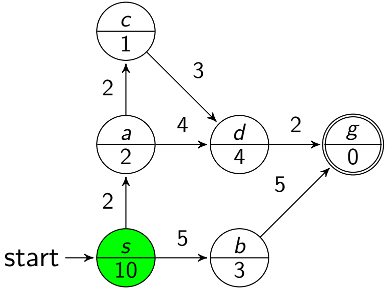
$Q \leftarrow \langle \texttt{start} \rangle$; // Initialize the queue with the starting node
while *Q is not empty* **do**
  Pick the path P with minimum heuristic cost $h(head(P))$ from the queue Q;
  if $head(P) =$ goal **then return** P ; // We have reached the goal
  foreach *vertex v such that* $(head(P), v) \in E$, **do**
    add $\langle v, P \rangle$ to the queue Q;

return FAILURE ; // Nothing left to consider.

Example of Greedy (Best-First) Search: Step 1

Q:

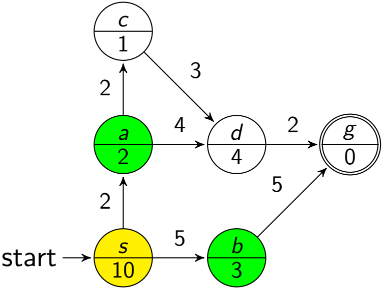
path	cost	h
$\langle s \rangle$	0	10



Example of Greedy (Best-First) Search: Step 2

Q:

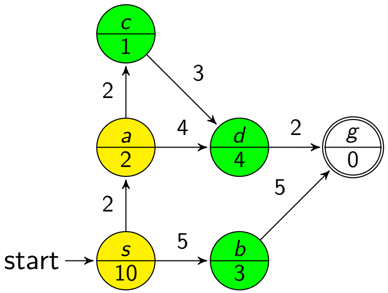
path	cost	h
$\langle a, s \rangle$	2	2
$\langle b, s \rangle$	5	3



Example of Greedy (Best-First) Search: Step 3

Q:

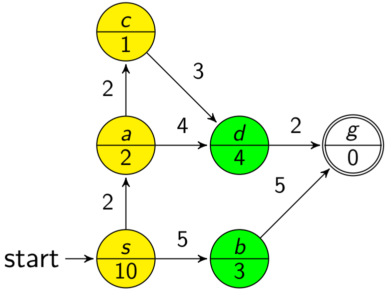
path	cost	h
$\langle c, a, s \rangle$	4	1
$\langle b, s \rangle$	5	3
$\langle d, a, s \rangle$	6	4



Example of Greedy (Best-First) Search: Step 4

Q:

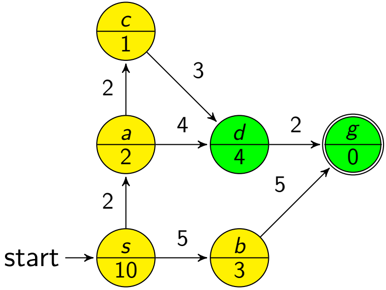
path	cost	h
$\langle b, s \rangle$	5	3
$\langle d, a, s \rangle$	6	4
$\langle d, c, a, s \rangle$	7	4



Example of Greedy (Best-First) Search: step 5

Q:

path	cost	h
$\langle g, b, s \rangle$	10	0
$\langle d, a, s \rangle$	6	4
$\langle d, c, a, s \rangle$	7	4



Remarks on Greedy (Best-First) Search

- Greedy (Best-First) search is similar in spirit to Depth-First Search: it keeps exploring until it has to back up due to a dead end.
- Greedy search is not complete and not optimal, but is often fast and efficient, depending on the heuristic function h.
- Worst-case time and space complexity $O(b^m)$.

Outline

1. Informed search methods: Introduction
2. Optimal search
   - A search
3. Dynamic Programming

The A search algorithm

The problems

- Uniform-Cost search is optimal, but may wander around a lot before finding the goal.
- Greedy search is not optimal, but in some cases it is efficient, as it is heavily biased towards moving towards the goal. The non-optimality comes from neglecting "the past."

The A search algorithm

The problems

- Uniform-Cost search is optimal, but may wander around a lot before finding the goal.
- Greedy search is not optimal, but in some cases it is efficient, as it is heavily biased towards moving towards the goal. The non-optimality comes from neglecting "the past."

The idea

- Keep track both of the cost of the partial path to get to a vertex, say $g(v)$, and of the heuristic function estimating the cost to reach the goal from a vertex, $h(v)$.
- In other words, choose as a "ranking" function the sum of the two costs:

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

  - $g(v)$: cost-to-come (from the start to v).
  - $h(v)$: cost-to-go estimate (from v to the goal).
  - $f(v)$: estimated cost of the path (from the start to v and then to the goal).

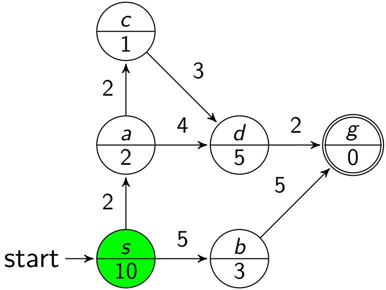
A Search

```
Q ← ⟨start⟩;            // Initialize the queue with the starting node
while Q is not empty do
    Pick the path P with minimum estimated cost f(P) = g(P) + h(head(P))
    from the queue Q;
    if head(P) = goal then return P ;        // We have reached the goal
    foreach vertex v such that (head(P), v) ∈ E, do
        add ⟨v, P⟩ to the queue Q;
return FAILURE ;                         // Nothing left to consider.
```

Example of A Search: Step 1

Q:

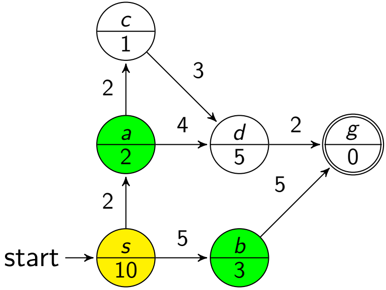
path	g	h	f
$\langle s \rangle$	0	10	10



Example of A Search: step 2

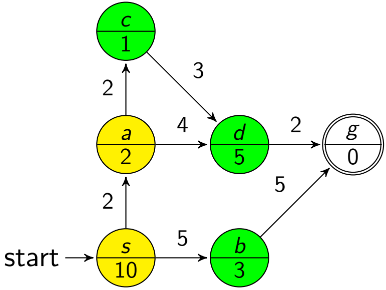
Q:

path	g	h	f
$\langle a, s \rangle$	2	2	4
$\langle b, s \rangle$	5	3	8



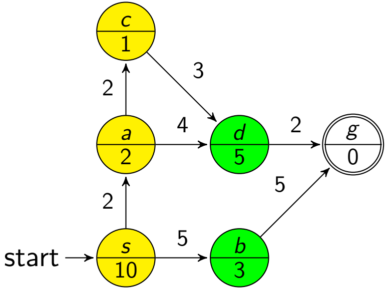
Example of A Search: step 3

	path	g	h	f
Q:	$\langle c, a, s \rangle$	4	1	5
	$\langle b, s \rangle$	5	3	8
	$\langle d, a, s \rangle$	6	5	11



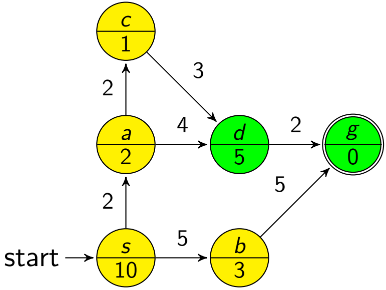
Example of A Search: step 4

	path	g	h	f
Q:	$\langle b, s \rangle$	5	3	8
	$\langle d, a, s \rangle$	6	5	11
	$\langle d, c, a, s \rangle$	7	5	12



Example of A Search: step 5

	path	g	h	f
Q:	$\langle g, b, s \rangle$	10	0	10
	$\langle d, a, s \rangle$	6	5	11
	$\langle d, c, a, s \rangle$	7	5	12



Remarks on the A search algorithm

- A search is similar to UCS, with a bias induced by the heuristic h. If $h = 0$, A $=$ UCS.
- The A search is complete, but is not optimal. What is wrong? *(Recall that if $h = 0$ then $A = UCS$, and hence optimal...)*

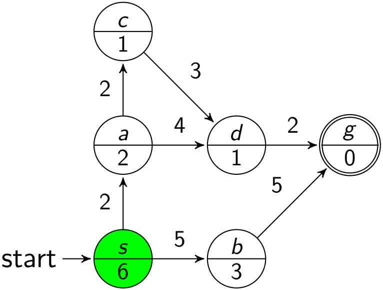
A^* Search

- Choose an admissible heuristic, i.e., such that $h(v) \leq h^*(v)$. (The star means "optimal.")
- The A search with an admissible heuristic is called A^*, which is guaranteed to be optimal.

Example of A^* Search: step 1

Q:

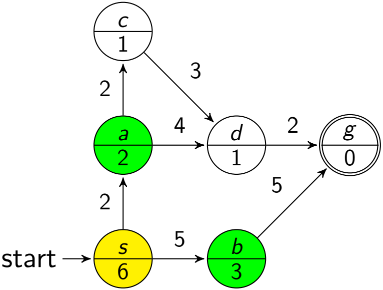
path	g	h	f
$\langle s \rangle$	0	10	10



Example of A^* Search: step 2

Q:

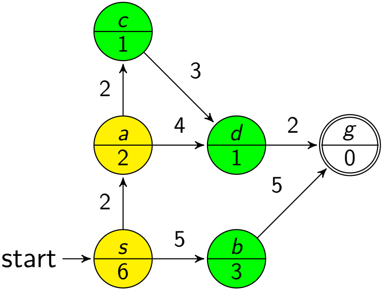
path	g	h	f
$\langle a, s \rangle$	2	2	4
$\langle b, s \rangle$	5	3	8



Example of A^* Search: step 3

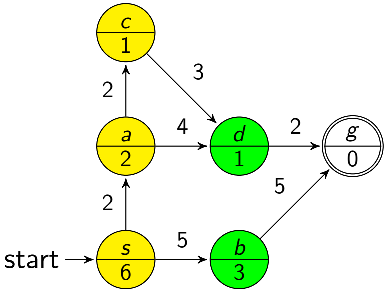
path	g	h	f
$\langle c, a, s \rangle$	4	1	5
$\langle d, a, s \rangle$	6	1	7
$\langle b, s \rangle$	5	3	8

Q:



Example of A^* Search: Step 4

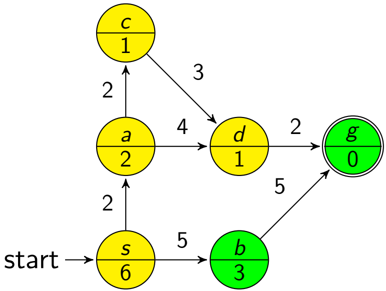
	path	g	h	f
Q:	$\langle d, a, s \rangle$	6	1	7
	$\langle b, s \rangle$	5	3	8
	$\langle d, c, a, s \rangle$	7	1	8



Example of A^* Search: step 5

Q:

path	g	h	f
$\langle g, d, a, s \rangle$	8	0	8
$\langle b, s \rangle$	5	3	8
$\langle d, c, a, s \rangle$	7	1	8



Proof (sketch) of A^* optimality

By contradiction

- Assume that A^* returns P, but $w(P) > w^*$ (w^* is the optimal path weight/cost).
- Find the first unexpanded node on the optimal path P^*, call it n.
- $f(n) > w(P)$, otherwise we would have expanded n.
- $f(n) = g(n) + h(n)$ by definition
- $\quad = g^*(n) + h(n)$ because n is on the optimal path.
- $\quad \leq g^*(n) + h^*(n)$ because h is admissible
- $\quad = f^*(n) = W^*$ because h is admissible
- Hence $W^* \geq f(n) > W$, which is a contradiction.

Admissible heuristics

How to find an admissible heuristic? i.e., a heuristic that never overestimates the cost-to-go.

Admissible heuristics

How to find an admissible heuristic? i.e., a heuristic that never overestimates the cost-to-go.

Examples of admissible heuristics

- $h(v) = 0$: this always works! However, it is not very useful, and in this case $A^* = UCS$.
- $h(v) = \mathrm{distance}(v, g)$ when the vertices of the graphs are physical locations.
- $h(v) = \|v - g\|_p$, when the vertices of the graph are points in a normed vector space.

Admissible heuristics

How to find an admissible heuristic? i.e., a heuristic that never overestimates the cost-to-go.

Examples of admissible heuristics

- $h(v) = 0$: this always works! However, it is not very useful, and in this case $A^* = UCS$.
- $h(v) = \mathrm{distance}(v, g)$ when the vertices of the graphs are physical locations.
- $h(v) = \|v - g\|_p$, when the vertices of the graph are points in a normed vector space.

A general method

Choose h as the optimal cost-to-go function for a relaxed problem, that is easy to compute.
(Relaxed problem: ignore some of the constraints in the original problem)

Admissible heuristics for the 8-puzzle

Initial state:

1		5
2	6	3
7	4	8

Goal state:

1	2	3
4	5	6
7	8	

Which of the following are admissible heuristics?

- $h = 0$
- $h = 1$
- $h =$ number of tiles in the wrong positon
- $h =$ sum of (Manhattan) distance between tiles and their goal position.

Admissible heuristics for the 8-puzzle

Initial state:

1		5
2	6	3
7	4	8

Goal state:

1	2	3
4	5	6
7	8	

Which of the following are admissible heuristics?

- $h = 0$ **YES**, always good
- $h = 1$
- $h =$ number of tiles in the wrong positon
- $h =$ sum of (Manhattan) distance between tiles and their goal position.

Admissible heuristics for the 8-puzzle

Initial state:

1		5
2	6	3
7	4	8

Goal state:

1	2	3
4	5	6
7	8	

Which of the following are admissible heuristics?

- $h = 0$ **YES**, always good
- $h = 1$ **NO**, not valid in goal state
- $h =$ number of tiles in the wrong positon **YES**, "teleport" each tile to the goal in one move
- $h =$ sum of (Manhattan) distance between tiles and their goal position.

Admissible heuristics for the 8-puzzle

Initial state:

1		5
2	6	3
7	4	8

Goal state:

1	2	3
4	5	6
7	8	

Which of the following are admissible heuristics?

- $h = 0$ **YES**, always good
- $h = 1$ **NO**, not valid in goal state
- $h =$ number of tiles in the wrong positon **YES**, "teleport" each tile to the goal in one move
- $h =$ sum of (Manhattan) distance between tiles and their goal position. **YES**, move each tile to the goal ignoring other tiles.

A partial order of heuristic functions

Some heuristics are better than others

- $h = 0$ is an admissible heuristic, but is not very useful.
- $h = h^*$ is also an admissible heuristic, and it the "best" possible one (it give us the optimal path directly, no searches/backtracking)

Partial order

- We say that h_1 dominates h_2 if $h_1(v) \geq h_2(v)$ for all vertices v.
- Clearly, h^* dominates all admissible heuristics, and 0 is dominated by all admissible heuristics.

Choosing the right heuristic

In general, we want a heuristic that is as close to h^* as possible. However, such a heuristic may be too complicated to compute. There is a tradeoff between complexity of computing h and the complexity of the search.

Consistent heuristics

- An additional useful property for A^* heuristics is called consistency
- A heuristic $h : X \to \mathbb{R}_{\geq 0}$ is said consistent if

$$h(u) \leq w\left(e = (u, v)\right) + h(v), \quad \forall (u, v) \in E.$$

- In other words, a consistent heuristics satisfies a triangle inequality.
- If h is a consistent heuristics, then $f = g + h$ is non-decreasing along paths:

$$f(v) = g(v) + h(v) = g(u) + w(u, v) + h(v) \geq f(u).$$

- Hence, the values of f on the sequence of nodes expanded by A^* is non-decreasing: the first path found to a node is also the optimal path $\Rightarrow$ no need to compare costs!

Outline

Dynamic Programming

The optimality principle

Let $P = (s, \ldots, v, \ldots g)$ be an optimal path (from s to g). Then, for any $v \in P$, the sub-path $S = (v, \ldots, g)$ is itself an optimal path (from v to g).

Using the optimality principle

- Essentially, optimal paths are made of optimal paths. Hence, we can construct long complex optimal paths by putting together short optimal paths, which can be easily computed.
- Fundamental formula in dynamic programming:

$$h^*(u) = \min_{(u,v) \in E} \left[w(\ (u,v)\ ) + h^*(v) \right].$$

- Typically, it is convenient to build optimal paths working backwards from the goal.

A special case of dynamic programming

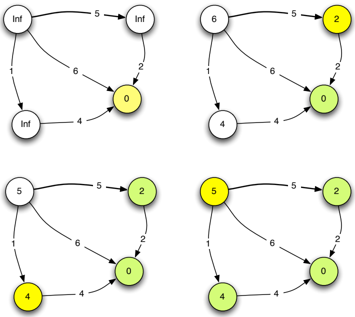
Dijkstra's algorithm

$Q \leftarrow V$ {All states get in the queue}.
for all $v \in V$, $\bar{h}(v) = (\infty$ if $v \in V_{\mathrm{G}}, 0$ otherwise)
while $Q \neq \emptyset$ **do**
  $u \leftarrow \arg\min_{v \in Q} \bar{h}(v)$ {Pick minimum-cost vertex in Q}
  for all $e = (v, u) \in E$ **do**
    $\bar{h}(v) \leftarrow \min\{\bar{h}(v), \bar{h}(u) + w(e)\}$ {Relax costs}

Recovering optimal paths

- The output of Dijkstra's algorithm is in fact the optimal cost-to-go function, h^*.
- From any vertex, we can compute the optimal outgoing edge via the dynamic programming equation.

Dijkstra's algorithm: example



- Dynamic programming requires the computation of all optimal sub-paths, from all possible initial states (curse of dimensionality).
- On-line computation is easy via state feedback: convert an open-loop problem into a feedback problem. This can be useful in real-world applications, where the state is subject to errors.

Concluding remarks

- A^* optimal and very effective in many situations. However, in some applications, it requires too much memory. Some possible approaches to address this problem include
  - Branch and bound
  - Conflict-directed A^*
  - Anytime A^*
- Other search algorithms
  - D^* and D^*-lite: versions of A^* for uncertain graphs.
  - Hill search: move to the neighboring state with the lowest cost.
  - Hill search with backup: move to the neighboring state with the lowest cost, keep track of unexplored states.
  - Beam algorithms: keep the best k partial paths in the queue.

16.410 / 16.413 Principles of Autonomy and Decision Making

Fall 2010