CS 4700: Foundations of Artificial Intelligence

Fall 2019

Prof. Haym Hirsh

Lecture 4

September 16, 2019

Iterative Deepening Search

```
IDS(s,ops)
        i=0
        loop
                result = DBDFS(s,ops,{},{},i)
                if result = fail then i=i+1
                else return(result)
```

Case 3b: Search Graph Using Graph Search

(Assuming finite state space,
maximum distance from initial state m)

Method	DFS	BFS	IDS
Complete?	Yes	Yes	Yes
Optimal?	No	Yes	Yes
Time	$O(b^m)$	$O(b^d)$	$O(b^d)$
Space	$O(bm)$	$O(b^d)$	$O(bd)$

Case 3b: Search Graph Using Graph Search

(Assuming finite state space,
maximum distance from initial state m)

Method	DFS	BFS	IDS
Complete?	Yes	Yes	Yes
Optimal?	No	Yes	Yes
Time	$O(b^m)$	$O(b^d)$	$O(b^d)$
Space	$O(bm)$	$O(b^d)$	$O(bd)$

Why?

Time Complexity of IDS

- Time for depth-bounded depth-first search to level i: $O(b^i)$

Time Complexity of IDS

- Time for depth-bounded depth-first search to level i: $O(b^i)$
- Time for IDS is the sum of this time for i = 0, …, d

Time Complexity of IDS

- Time for depth-bounded depth-first search to level i: $O(b^i)$
- Time for IDS is the sum of this time for i = 0, …, d

$$\sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1}$$

Time Complexity of IDS

- Time for depth-bounded depth-first search to level i: $O(b^i)$
- Time for IDS is the sum of this time for i = 0, …, d

$$\sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1}$$

This is $O(b^d)$

Time Complexity of IDS

- Time for depth-bounded depth-first search to level i: $O(b^i)$
- Time for IDS is the sum of this time for i = 0, …, d

$$\sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1}$$

This is $O(b^d)$

(You should be able to do this)

“Uninformed” Search

DFS

BFS

DBDFS

IDS

All you have is the structure of the space:

States, Initial State, Operators, Goal

“Informed” Search

- To formulate a problem:
  - States: S
  - Operators: Ops
  - Initial state
  - Goal condition: goal(s)
  - Heuristic Evaluation Function $f(s): S \rightarrow \mathbb{R}$ (usually ≥ 0)
    - f is an estimate of the merit of s
    - Typically $f(s_1) < f(s_2)$ means s_1 is “better” than s_2

Best-First Search

```
BestFS(s,ops,open,closed) =
        If goal(s) Then return(s);
        Else If not(s ∈ closed)
          Then
                successors ← {}; add(s,closed);
                For each o ∈ ops that applies to s
                        add apply(o,s) to successors
                open ← add successors to open;
        If not(empty(open))
                s' ← min_f(open);
                open ← remove(s',open);
                BestFS(s',ops,open,closed)
        `   Else return(FAIL)
```

Initial call: BestFS(initialstate,ops,{},{})

Best-First Search Generalizes Other Search Methods

- Given uniform operator costs
  - If f(s) = distance(s,initialstate), get Best-First Search
  - If f(s) inverts this (for example, - distance(s,initialstate) or 1/distance(s,initialstate)), get Depth-First Search
- Non-uniform operator costs
  - If f(s) = sum of costs from initial state to s

Best-First Search Generalizes Other Search Methods

- Given uniform operator costs
  - If f(s) = distance(s,initialstate), get Best-First Search
  - If f(s) inverts this (for example, - distance(s,initialstate) or 1/distance(s,initialstate)), get Depth-First Search
- Non-uniform operator costs
  - If f(s) = sum of costs from initial state to s

Uniform Cost Search, aka

Dijkstra's Algorithm

Best-First Search Generalizes Other Search Methods

- Given uniform operator costs
  - If f(s) = distance(s,initialstate), get Best-First Search
  - If f(s) inverts this (for example, - distance(s,initialstate) or 1/distance(s,initialstate)), get Depth-First Search
- Non-uniform operator costs
  - If f(s) = sum of costs from initial state to s

    Uniform Cost Search, aka

    Dijkstra's Algorithm
  - Often use g(s) instead of f(s) for this special case

Best-First Search Generalizes Other Search Methods

- Given uniform operator costs
  - If f(s) = distance(s,initialstate), get Best-First Search
  - If f(s) inverts this (for example, - distance(s,initialstate) or 1/distance(s,initialstate)), get Depth-First Search
- Non-uniform operator costs
  - If f(s) = sum of costs from initial state to s

    Uniform Cost Search, aka

    Dijkstra's Algorithm
  - Often use g(s) instead of f(s) for this special case

I said f(s) estimates the quality of s – but there's no estimation here ????

A* Search

- Special case of best-first search with a specific form of evaluation function:

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

where

- $g(s)$ is the cost of getting to s (same as before)
- $h(s)$ is an estimate of the cost getting to a goal state

A* Search

- Special case of best-first search with a specific form of evaluation function:

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

where

- $g(s)$ is the cost of getting to s (same as before)
- $h(s)$ is an estimate of the cost getting to a goal state

$f(s)$ is an estimate of the cost of getting from the initial state to a goal state going through s

A* Search

- Special case of best-first search with a specific form of evaluation function:

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

where

- g(s) is the cost of getting to s (same as before)
- h(s) is an estimate of the cost getting to a goal state

f(s) is an estimate of the cost of getting from the initial state to a goal state going through s

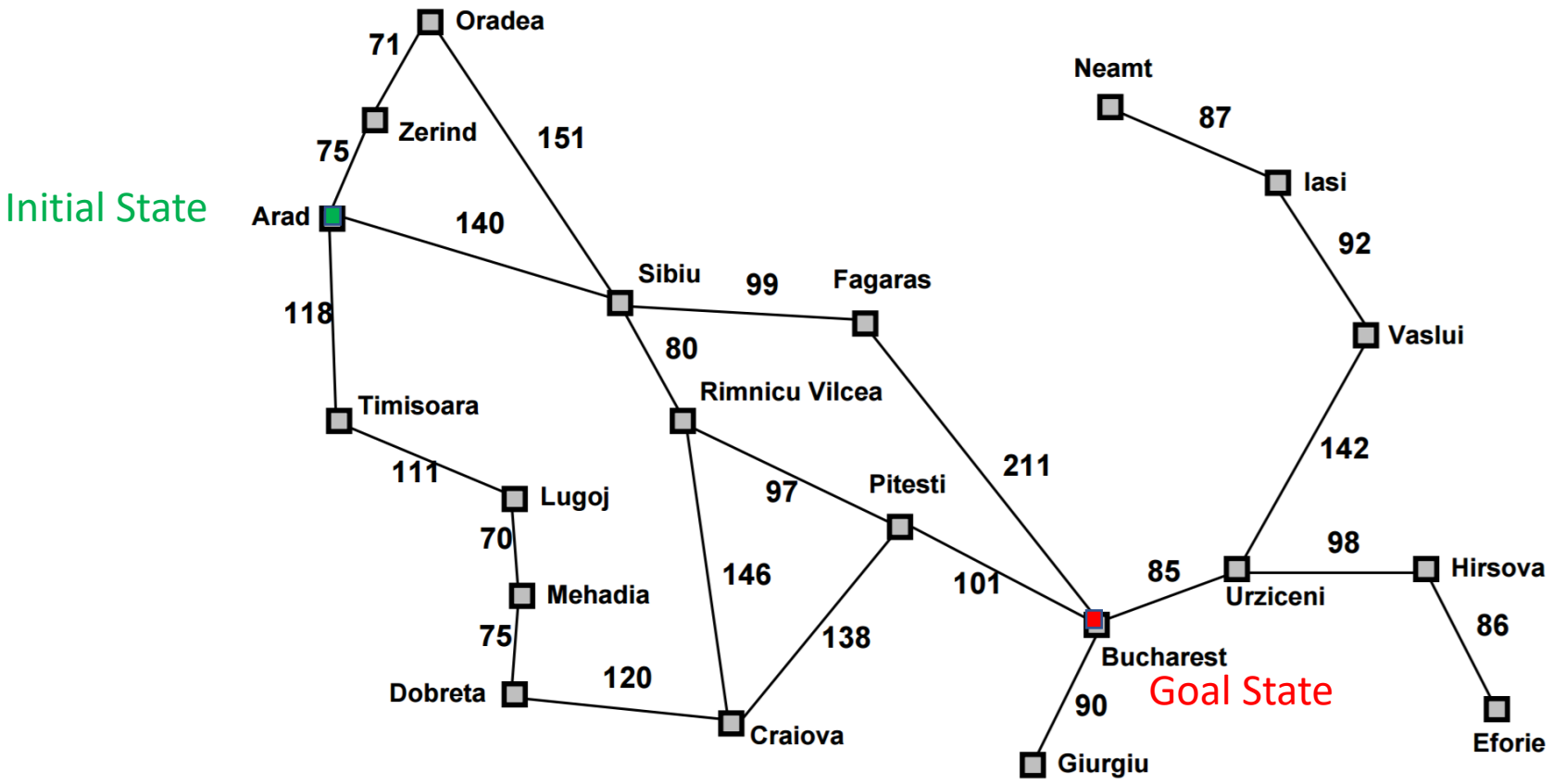
(Warning: sometimes “heuristic evaluation function” is used to refer just to h(s) and sometimes to all of f(s) – they’re both “estimates”)

Why f(s) = g(s) + h(s)?

Answer 1:

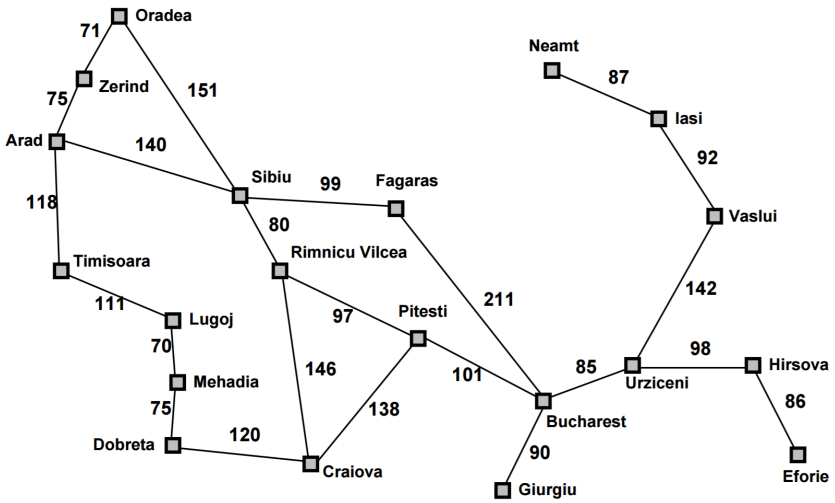
For many problems it isn’t too difficult to come up with an h(s)

Heuristic Evaluation Function: Route Finding



Heuristic Evaluation Function: Route Finding

- g(s) = sum of costs from Arad to a city s
- h(s) = distance between s and Bucharest on a map
- f(s) = estimate of distance from Arad to Bucharest through s



h(s)=distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Heuristic Evaluation Function: 15 puzzle

4	1	2	3
5	6	7	11
8	9		10
12	13	14	15

$\rightarrow$

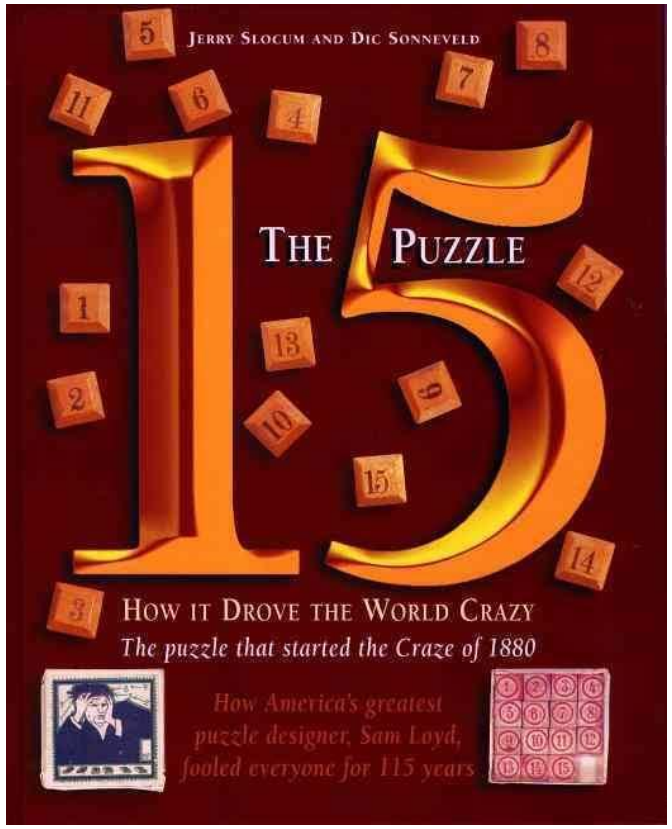
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Heuristic Evaluation Function: 15 puzzle

4	1	2	3
5	6	7	11
8	9		10
12	13	14	15

→

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	



Heuristic Evaluation Function: 15 puzzle

4	1	2	3
5	6	7	11
8	9		10
12	13	14	15

$\rightarrow$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Heuristic Evaluation Function: 15 puzzle

- g(s) = number of moves to get to s
- h(s) = number of out of place numbers in s

4	1	2	3
5	6	7	11
8	9		10
12	13	14	15

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

- f(s) = estimate of solution length from initial state to goal via s

Heuristic Evaluation Function: 15 puzzle

- g(s) = number of moves to get to s
- h(s) = number of out of place numbers in s

h(s) = 12

4	1	2	3
5	6	7	11
8	9		10
12	13	14	15

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

- f(s) = estimate of solution length from initial state to goal via s

Heuristic Evaluation Function: 15 puzzle

- g(s) = number of moves to get to s
- h(s) = sum of distances of each number from its correct position

4	1	2	3
5	6	7	11
8	9		10
12	13	14	15

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

- f(s) = estimate of solution length from initial state to goal via s

Heuristic Evaluation Function: 15 puzzle

- g(s) = number of moves to get to s
- h(s) = sum of distances of each number from its correct position

h(s) = 21

4	1	2	3
5	6	7	11
8	9		10
12	13	14	15

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

- f(s) = estimate of solution length from initial state to goal via s

Heuristic Evaluation Function: 15 puzzle

- g(s) = number of moves to get to s
- h(s) = sum of distances of each number from its correct position

h(s) = 21

4	1	2	3
5	6	7	11
8	9		10
12	13	14	15

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

"Manhattan Distance"

- f(s) = estimate of solution length from initial state to goal via s

Common Property of These Two h(s) Examples

Denote the actual cost of an optimal path to a goal state $h^*(s)$

Common Property of These Two h(s) Examples

Denote the actual cost of an optimal path to a goal state $h^*(s)$

$$0 \leq h(s) \leq h^*(s)$$

Common Property of These Two h(s) Examples

Denote the actual cost of an optimal path to a goal state $h^*(s)$

$$0 \leq h(s) \leq h^*(s)$$

Map distance never over-estimates driving distance

Manhattan distance never over-estimates solution distance

Admissible Heuristic Functions

- A function h(s) is “admissible” if for every state s:

$$0 \leq h(s) \leq h^*(s)$$

Why f(s) = g(s) + h(s)?

Answer 1:

For many problems it isn’t too difficult to come up with an h(s)

Why f(s) = g(s) + h(s)?

Answer 2:

Knowing something about h(s) can
give you guarantees on the outcome of search

Admissible Heuristic Functions

- A function h(s) is “admissible” if for every state s:

$$0 \leq h(s) \leq h^*(s)$$

- If h(s) is admissible then A* search is guaranteed to find an optimal solution

Admissible Heuristic Functions

- A function h(s) is “admissible” if for every state s:

$$0 \leq h(s) \leq h^*(s)$$

Best-first search with f(s) = g(s) + h(s)

- If h(s) is admissible then A* search is guaranteed to find an optimal solution

Admissible Heuristic Functions

- A function h(s) is "admissible" if for every state s:

$$0 \leq h(s) \leq h^*(s)$$

Best-first search with f(s) = g(s) + h(s)

- If h(s) is admissible then A* search is guaranteed to find an optimal solution

(need slight modification to Best-First Search)

Search Algorithm Template

```
Search(s,ops,open,closed) =
        If goal(s) Then return(s);
        Else If not(s ∈ closed)
          Then
                successors ← {}; add(s,closed);
                For each o ∈ ops that applies to s
                        add apply(o,s) to successors
                open ← add successors to open;
        If not(empty(open))
                s' ← select(open);
                open ← remove(s',open);
                search(s',ops,open,closed)
        `   Else return(FAIL)
```

Search Algorithm Template

```
Search(s,ops,open,closed) =
        If goal(s) Then return(s);
        Else If not(s ∈ closed)
          Then
                successors ← {}; add(s,closed);
                For each o ∈ ops that applies to s
                        add apply(o,s) to successors
                open ← add successors that are not already in open to open;
        If not(empty(open))
                s' ← select(open);
                open ← remove(s',open);
                search(s',ops,open,closed)
        `   Else return(FAIL)
```

A* Search

```
Search(s,ops,open,closed) =
        If goal(s) Then return(s);
        Else If not(s ∈ closed)
          Then
                successors ← {}; add(s,closed);
                For each o ∈ ops that applies to s
                        add apply(o,s) to successors
                open ← add successors that are not already in open to open;
        If not(empty(open))        If a successor is in open and the new g(s) is smaller update g(s) in open
                s' ← select(open);
                open ← remove(s',open);
                search(s',ops,open,closed)
        `   Else return(FAIL)
```

A* Search

```
Search(s,ops,open,closed) =
        If goal(s) Then return(s);
        Else If not(s ∈ closed)
          Then
                successors ← {}; add(s,closed);
                For each o ∈ ops that applies to s
                        add apply(o,s) to successors
                open ← add successors that are not already in open to open;
        If not(empty(open))
                s' ← select(open);
                open ← remove(s',open);
                search(s',ops,open,closed)
        `   Else return(FAIL)
```

If a successor is in open and the new g(s) is smaller update g(s) in open

If a successor is in closed and the new g(s) is smaller, put s back on open

A* Search

- If a successor is in open and the new g(s) is smaller:

  Handles multiple paths to the same state

- If a successor is in closed and the new g(s) is smaller:

  Necessary to guarantee optimality

Admissible Heuristic Functions

- A function h(s) is “admissible” if for every state s:

$$0 \leq h(s) \leq h^*(s)$$

- If h(s) is admissible then A* search is guaranteed to find an optimal solution

WHY?

Admissible Heuristic Functions

- A function h(s) is “admissible” if for every state s:

$$0 \leq h(s) \leq h^*(s)$$

- If h(s) is admissible then A* search is guaranteed to find an optimal solution

WHY?

You would never take off a goal state if a better state is in open

A* Search

- If a successor is in open and the new g(s) is smaller:

  Handles multiple paths to the same state

- If a successor is in closed and the new g(s) is smaller:

  Necessary to guarantee optimality

A* Search

- If a successor is in open and the new g(s) is smaller:

  Handles multiple paths to the same state

- If a successor is in closed and the new g(s) is smaller:

  Necessary to guarantee optimality

  s might have been explored further,
  need to now follow that through

Consistent Evaluation Functions

- A function h(s) is “consistent” (or “monotonic”) if it satisfies the triangle inequality for all s:

$$h(s) \leq h(apply(a,s)) + cost(apply(a,s))$$

Intuitively: the estimate gets increasingly better over time

Consistency vs Admissibility

- Consistency is stronger than admissibility:

  If h(s) is consistent then h(s) is admissible

- Consistency lets you improve the A* algorithm

A* Search

```
Search(s,ops,open,closed) =
        If goal(s) Then return(s);
        Else If not(s ∈ closed)
          Then
                successors ← {}; add(s,closed);
                For each o ∈ ops that applies to s
                        add apply(o,s) to successors
                open ← add successors that are not already in open to open;
        If not(empty(open))
                s' ← select(open);
                open ← remove(s',open);
                search(s',ops,open,closed)
        `   Else return(FAIL)
```

If a successor is in open and the new g(s) is smaller update g(s) in open

If a successor is in closed and the new g(s) is smaller, put s back on open