Heuristic Search

Blai Bonet

Universidad Simón Bolívar, Caracas, Venezuela

Heuristics

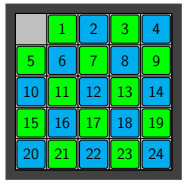
Goals for the lecture

- We cannot solve huge problems with blind (brute force) search
- Need information and new algorithms
- Heuristics as source of information
- Examples
- Properties of heuristics

Challenges

8-puzzle

15-puzzle



24-puzzle

$2\times2\times2$

$3\times3\times3$

family

Brute-force search algorithms

A **brute-force** search algorithm explores the search tree looking for a goal state **without any additional information about the problem**

All algorithms seen so far are brute-force search algorithms:

- breadth-first search and depth-first search
- uniform-cost search
- iterative deepening depth-first search
- iterative deepening uniform-cost search

A brute-force algorithm cannot **overcome the branching factor** of the search tree (i.e. the combinatorial explosion)

The need for heuristics

problem	#nodes	time for brute-force search (10 million nodes/second)
8-puzzle	10^5	0.01 seconds
$2\times2\times2$ Rubik's cube	10^6	0.2 seconds
15-puzzle	10^{13}	6 days
$3\times3\times3$ Rubik's cube	10^{19}	68,000 years
24-puzzle	10^{25}	12 billion years

Data from [Korf, Invited talk at SARA-2000]

What is a heuristic?

In **heuristic search**, a heuristic is a **function** that provides **estimates** on the **minimum-cost** to reach a goal state for a given state

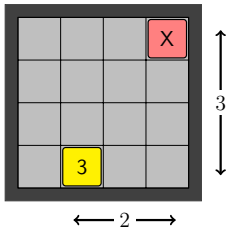
Formally, a heuristic h is a function $h : S \to \mathbb{R}^{\geq 0} \cup \{\infty\}$ that maps states to non-negative reals plus infinity

- $h(s)$ is the estimate for state s provided by the heuristic h
- $h(s) = \infty$ means that the heuristic h **"thinks"** that a **goal state can't be reached** from state s

15-puzzle: Manhattan-distance heuristic

The Manhattan distance between two positions (x, y) and (x', y') in a rectangular grid is $|x - x'| + |y - y'|$

The Manhattan distance for a tile in the 15-puzzle is the Manhattan distance between the tile's position and its goal position



Manhattan distance for tile 3 is $2 + 3 = 5$

15-puzzle: Manhattan-distance heuristic

The Manhattan distance between two positions (x, y) and (x', y') in a rectangular grid is $|x - x'| + |y - y'|$

The Manhattan distance for a tile in the 15-puzzle is the Manhattan distance between the tile's position and its goal position

The Manhattan-distance heuristic maps a state into the **sum of the Manhattan distances** for each tile in the state

$1+2+1+4+2+2+2+1+2+2+2+1+2+2+1 = \mathbf{26}$

Romania: straigth-line distances to Bucharest

A rough estimate on the distance to go from a given city to Bucharest is the **straight line** or **euclidean** distance

city	sl-distance	city	sl-distance
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

TSP: search tree and partial tours

Recall the TSP problem where the task is to find a minimum cost tour over a set $C = \{c_1, c_2, \ldots, c_n\}$ of n cities

Consider a search tree that "grows" a partial tour from a given city a_1 until it becomes a complete tour

Each node in the search tree corresponds to a partial tour of the form $\langle a_1, a_2, \ldots, a_k \rangle$

Given such partial tour, a heuristic for the TSP (for this search tree) must estimate the **cost of completing the tour** over all cities in C

TSP: tour completion and Hamiltonian paths

For completing a partial tour $\langle a_1, a_2, \ldots, a_k \rangle$ over the cities $\{a_1, \ldots, a_k\}$ with **extreme points** a_1 and a_k, we need a path that:

- starts at the city a_1
- ends at the city a_k
- visits each city in $C' = C \setminus \{a_2, \ldots, a_{k-1}\}$ **exactly once**

Such a path is called a **Hamiltonian path** from a_1 to a_k over C'

Therefore, a best completion of the partial tour corresponds to a **minimum-cost Hamiltonian path** from a_1 to a_k over C'

Unfortunately, computing a Hamiltonian path is **NP-hard** (i.e. intractable) and we don't know how to do it efficiently

TSP: MST heuristic

Observe that a path from a_1 to a_k over C' is a **special type of tree** (i.e. an undirected, connected and acyclic graph)

Therefore, the cost of minimum-cost tree over the cities in C' is an **estimate on the cost** of a minimum-cost Hamiltonian path from a_1 to a_k over C'

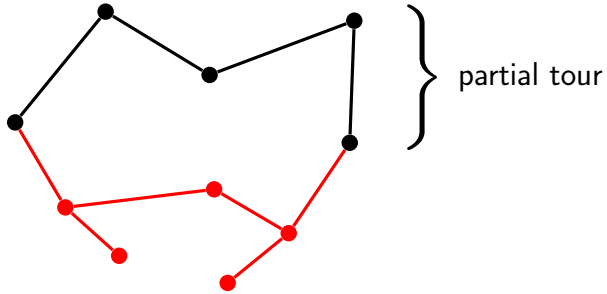
Such tree is a **minimum-weight spaning tree (MST)** over C'. It can be computed efficiently with Kruskal's or Prim's algorithm [CLRS]

Given a partial tour $\langle a_1, \ldots, a_k \rangle$, the MST heuristic is computed as:

1. Compute a MST T over $C \setminus \{a_1, \ldots, a_k\}$ with cost $c(T)$
2. Let e_1 and e_k be two edges of minimum cost that connect vertices a_1 and a_k to T respectively
3. The MST heuristic is $c(T) + c(e_1) + c(e_k)$

TSP: example of MST heuristic

TSP instance with 10 cities:



partial tour + MST

Properties for heuristics

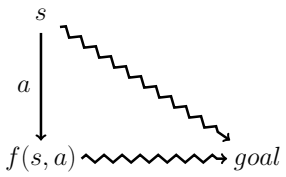
Properties for heuristic $h : S \to \mathbb{R}^{\geq 0} \cup \{\infty\}$:

Safe: $h(s) = \infty \implies$ there is no goal reachable from s

Goal-aware: s is a goal state $\implies h(s) = 0$

Admissible: $h(s)$ is **lower bound** on cost for reaching goal from s

Consistent: refers to **triangular inequality** among estimates



$$h(s) \leq c(s,a) + h(f(s,a))$$

Relations between properties for heuristics

Let $P = \langle S, A, s_{init}, S_G, f, c \rangle$ be a state-space model and $h : S \to \mathbb{R}^{\geq 0} \cup \{\infty\}$ be a heuristic function:

– if h is admissible, then h is safe

– if h is admissible, then h is goal-aware

– if h is goal-aware and consistent, then h is admissible

Proof of 1st: Let s be state with $h(s) = \infty$

We need to show that there is no path from s to a goal state

Since h is admissible, $h(s)$ is a lower bound on the cost of any path from s to a goal state

Therefore, since $h(s) = \infty$, there is no such path □

Relations between properties for heuristics

Let $P = \langle S, A, s_{init}, S_G, f, c \rangle$ be a state-space model and $h : S \to \mathbb{R}^{\geq 0} \cup \{\infty\}$ be a heuristic function:

- if h is admissible, then h is safe
- if h is admissible, then h is goal-aware
- if h is goal-aware and consistent, then h is admissible

Proof of 2nd: Let s be a goal state

We need to show that $h(s) = 0$

Since h is admissible, $h(s)$ is a lower bound on the cost of any path from s to a goal node

Since s is a goal state, the **empty path** is such a path and $h(s) \leq 0$

Since h is non-negative, $h(s) \geq 0$. Therefore, $h(s) = 0$ □

Relations between properties for heuristics

Let $P = \langle S, A, s_{init}, S_G, f, c \rangle$ be a state-space model and $h : S \to \mathbb{R}^{\geq 0} \cup \{\infty\}$ be a heuristic function:

- if h is admissible, then h is safe
- if h is admissible, then h is goal-aware
- if h is goal-aware and consistent, then h is admissible

Proof of 3rd: Let s be a state

We need to show that $h(s)$ is a lower bound on cost of any path from s to a goal state

If there is no path from s to goal, then $h(s)$ (whatever value) is a lower bound on cost of such path which is ∞

Relations between properties for heuristics

Let $P = \langle S, A, s_{init}, S_G, f, c \rangle$ be a state-space model and $h : S \to \mathbb{R}^{\geq 0} \cup \{\infty\}$ be a heuristic function:

- if h is admissible, then h is safe
- if h is admissible, then h is goal-aware
- if h is goal-aware and consistent, then h is admissible

Proof of 3rd: Let s be a state

We need to show that $h(s)$ is a lower bound on cost of any path from s to a goal state

Else, let $\langle s_0 = s, a_0, s_1, \ldots, s_n \rangle$ be **path from s to a goal** with cost c:

$$h(s) \leq c(s, a_0) + h(s_1) \leq c(s, a_0) + c(s_1, a_1) + h(s_2) \leq \cdots \leq c + h(s_n)$$

Since $h(s_n) = 0$, then $h(s) \leq c$. Therefore, h is admissible □

Summary

- Brute-force search is impractical for large spaces
- Heuristics provide information as estimates on cost to reach goal
- Examples: Manhattan distance, straight-line distance, MST
- Properties of heuristics and relation between them

Best-first search and A*

Goals for the lecture

- Three approaches for using the information provided by heuristics
- Greedy best-first search
- Best-first search
- A* = best-first search + admissible heuristic

What to do with the heuristic information?

Heuristic functions provide **guidance information** to reach the goal

Three approaches for using information:

– Consider the heuristic as perfect estimate (hill climbing: later)

– Use it to order nodes for expansion (greedy best-first search)

– Combine with information gathered during search (best-first search)

Greedy best-first search

GBFS explores the search tree by expanding nodes with less heuristic value first because they appear to be **closer to the goal**

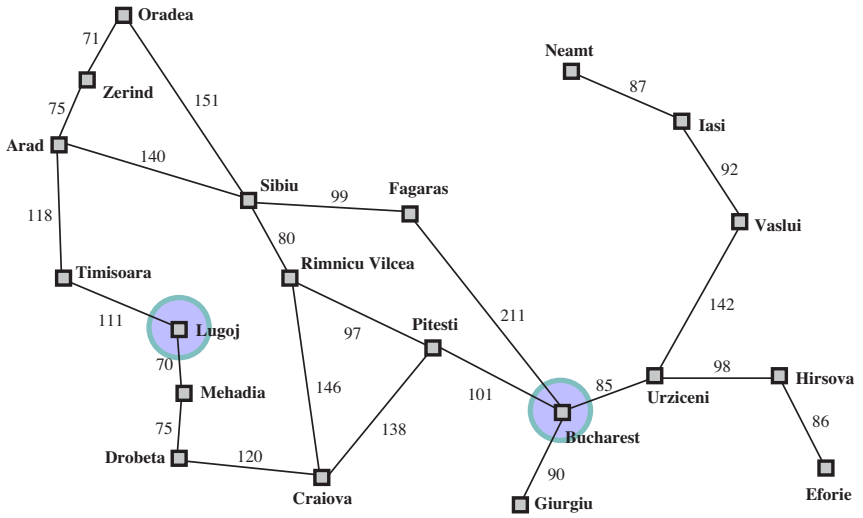
Nodes ordered for expansion using a **priority queue**

Algorithm can be implemented as a tree- or graph-search algorithm depending on whether duplicates are pruned or not

Greedy best-first search: pseudocode

```
1  % Greedy-best-first search with duplicate elimination
2  greedy-best-first-search(Heuristic h):
3      % initialization
4      PriorityQueue q                % ordered by heuristic value: h(s)
5      set-color(init(),Gray)
6      q.insert(make-root-node(init()), h(init()))
7
8      % search
9      while !q.empty()
10         Node n = q.pop()
11
12         % check for goal
13         if n.state.is-goal() return n
14
15         % expand node
16         foreach <s,a> in n.state.successors()
17             if get-color(s) == White && h(s) < ∞   % assumes safe h
18                 set-color(s,Gray)
19                 q.insert(n.make-node(s,a), h(s))
20         set-color(n.state,Black)
21
22     return null          % failure: there is no path from root to goal
```


Example: traveling in Romania [AIMA]

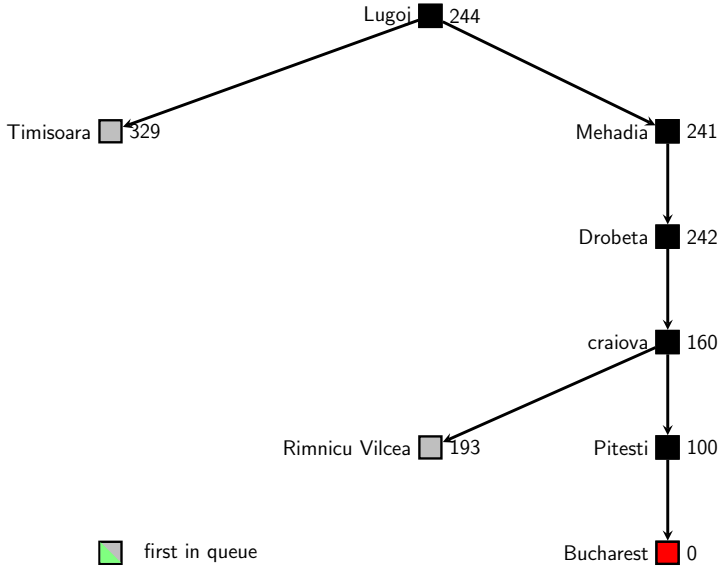


Task: find optimal path from **Lugoj** to **Bucharest**

Travelleing in Romania: straigth-line distances to Bucharest

city	sl-distance	city	sl-distance
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Greedy best-first search: example



Properties of greedy best-first search

- **Complete:** it is complete with duplicate elimination, without duplicate elimination it can loop forever
- **Optimality:** it isn't optimal as it may return **suboptimal** paths
- **Time complexity:** with duplicate elimination, complexity is $O(S)$
- **Space complexity:** with duplicate elimination, complexity is $O(S)$

where S is the state space of the model

Algorithmic challenge

How to exploit heuristic information while guaranteeing optimality?

Can it be done for any heuristic?

Should we require some property on the heuristic?

Best-first search

Best-first search combines the heuristic value with the cost of the path leading to the node to obtain a **cost estimate for the entire path**

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

This estimate is called the f-value of the node

Nodes ordered for expansion using a **priority queue** on f-values

Algorithm can be implemented as a tree- or graph-search algorithm depending on whether duplicates are pruned or not

Best-first search: pseudocode

```
1  % Best-first search with duplicate elimination
2  best-first-search(Heuristic h):
3      % initialization
4      PriorityQueue q              % ordered by f-value: f(n) = g(n) + h(s)
5      set-color(init(), Gray)
6      set-distance(init(), 0)
7      q.insert(make-root-node(init()), h(init()))
8
9      % search
10     while !q.empty()
11         Node n = q.pop()
12
13         % check for goal
14         if n.state.is-goal() return n
15
16         % expand node
17         best-first-search-expansion(n, q)
18         set-color(n.state, Black)
19
20     return null        % failure: there is no path from root to goal
```

Best-first search: pseudocode for expansion

```
21 Node best-first-search-expansion(Node n, PriorityQueue q):
22     foreach <s,a> in n.state.successors()
23         if h(s) == ∞ continue              % assumes safe heuristic
24         g = n.g + c(n.state, a)
25
26         % first time encountered
27         if get-color(s) == White
28             set-color(s, Gray)
29             set-distance(s, g)
30             q.insert(n.make-node(s, a), g + h(s))
31
32         % a shorter path was found
33         else if g < get-distance(s)
34             set-distance(s, g)
35             if get-color(s) == Gray   % re-order and re-link parent
36                 s.node.parent = n
37                 s.node.action = a
38                 s.node.g = g
39                 q.decrease-priority(s.node, g+h(s))
40             else
41                 % node is black: re-open state!
42                 set-color(s, Gray)
43                 q.insert(n.make-node(s, a), g + h(s))
```


Best-first search: notes on pseudocode

Need map states to path costs (upper bound on optimal-path costs):

- `get-distance(State s, unsigned d)`
- `set-distance(State s, unsigned d)`

For re-ordering nodes in the open list (i.e. priority queue):

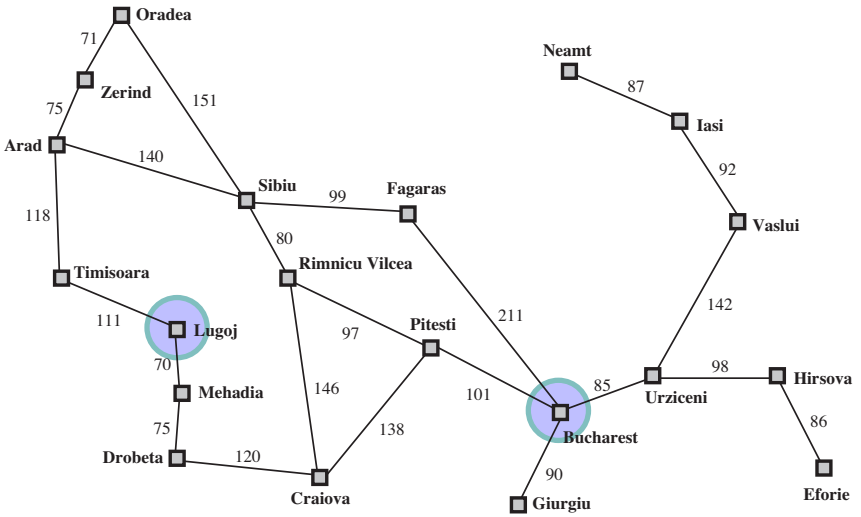
- Priority queue must implement **decrease priority** operation
- Need to link states to nodes:
  - Add `node` field to `AbstractState`
  - Invariants:

    `get-color(s) == Gray` $\iff$ `s.node` points to node in open list

    `get-color(s) == Gray` $\iff$ `s.node.state == s`

    `get-color(s) != Gray` $\iff$ `s.node == null`

Example: traveling in Romania [AIMA]

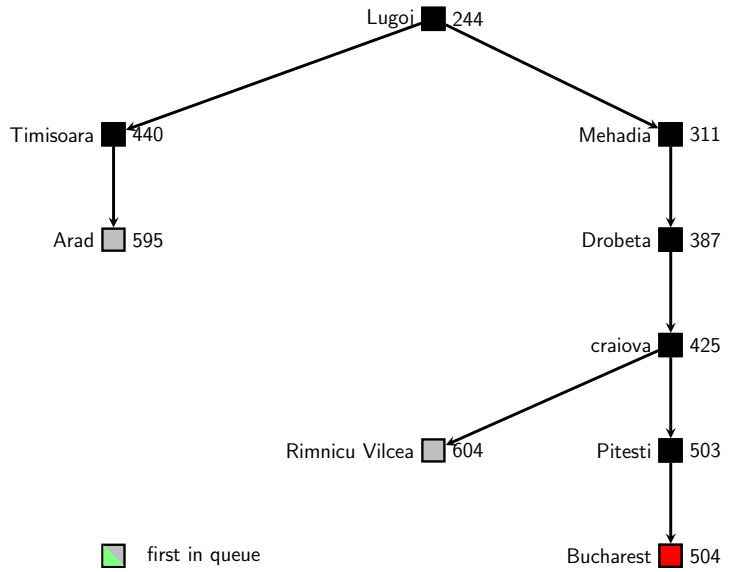


Task: find optimal path from **Lugoj** to **Bucharest**

Travelling in Romania: straigth-line distances to Bucharest

city	sl-distance	city	sl-distance
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Best-first search: example



Properties of best-first search

Best-first search with $h = 0$ is equal to uniform-cost search

- **Complete:** if there is a path, it finds the path (if duplicates pruned; partially complete if $c_{\min} > 0$)
- **Optimality:** returns optimal path if **heuristic is admissible**
- **Time complexity:** $O(b^{c^*/c_{\min}})$ $(= O(b^d)$ if all costs are 1)
- **Space complexity:** $O(b^{c^*/c_{\min}})$ $(= O(b^d)$ if all costs are 1)

If duplicates aren't pruned and minimum edge cost $c_{\min} = 0$, best-first search may not terminate (i.e. it's not complete)

Time and space complexities calculated in **canonical search tree** with branching factor b and $c_{\min} > 0$ where **minimum-cost** goal appears at depth d with cost c^*

A* = best-first search + admissible heuristic

The A* algorithm [Hart, Nilsson and Raphael, 1968] is best-first search with **admissible heuristic**

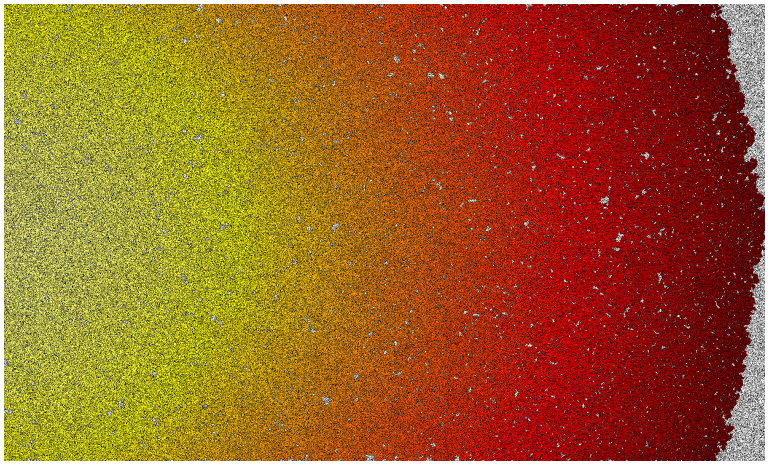
It is a fundamental algorithm in AI and CS

If the heuristic is not only admissible but also consistent, then A* is **asymptotically optimal**; i.e. no other algorithm using the same information is better (up to a constant factor) than A*

Comparison uniform-cost search vs. A*

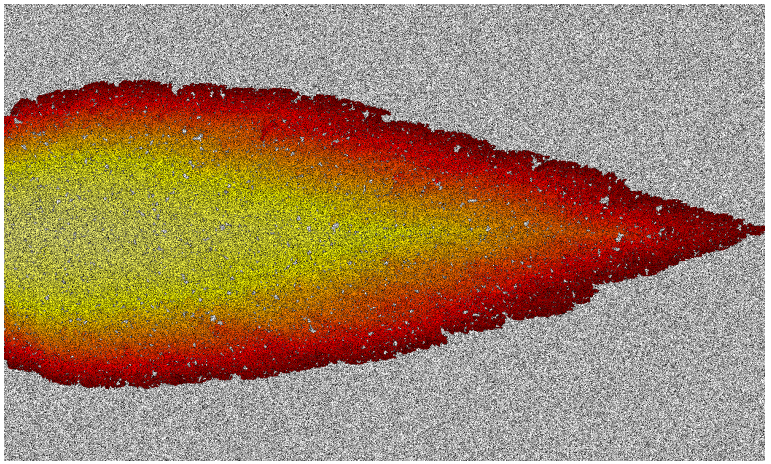
- Problem is a path-finding task in a 4-connected grid
- Heuristic is Manhattan distance assuming no obstacles
- Initial cell at the middle row and leftmost column
- Goal cell is at the middle row and rightmost column
- Colors for cells:
  - black cells are obstacles (i.e. non-traversable cells)
  - white cells were not expanded in the search
  - yellow cells were expanded early in the search
  - dark red cells were expanded late in the search

Example: uniform-cost search



[Image by Jordan Thayer. Http://www.jordanthayer.com]

Example: A*



[Image by Jordan Thayer. Http://www.jordanthayer.com]

Best-first search with delayed duplicate elimination

```
1  % best-first-search with delayed duplicate elimination
2  best-first-search(Heuristic h):
3      PriorityQueue q              % ordered by f-value: f(n) = g(n) + h(s)
4      set-color(init(), Gray)
5      q.insert(make-root-node(init()), h(init()))
6
7      while !q.empty()
8          Node n = q.pop()
9          State ns = n.state
10
11         % delayed duplicate elimination
12         if n.g < get-distance(ns)
13             set-distance(ns, n.g)
14             if ns.is-goal() return n
15
16             % expand node
17             foreach <s,a> in ns.successors()
18                 if h(s) < ∞
19                     set-color(s, Gray)
20                     q.insert(n.make-node(s, a), n.g + c(n.states,a) + h(s))
21             set-color(ns,Black)
22     return null        % failure: there is no path from root to goal
```


Best-first search with consistent heuristic

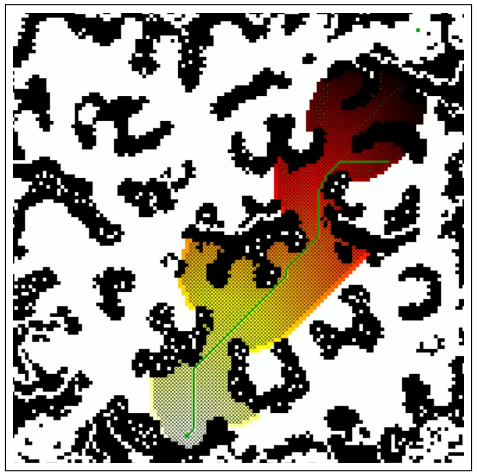
The case of consistent heuristics is very important because:

- best-first search can be simplified; no need for `decrease-key`
- best-first search yields stronger guarantees
- many heuristics are consistent

Indeed,

- No state is re-opened (but nodes may be re-ordered in OPEN)
- Every time a node is pulled out from the queue, **the path from the initial state to the node's state is optimal**

A* on a pathfinding problem (video)



[Video by Jordan Thayer. Http://www.jordanthayer.com]

Summary

- How to use information provided by heuristic
- Greedy best-first search: nodes ordered by h-values
- Best-first search: nodes ordered by f-values
- Guarantees offered by algorithms
- A* = best-first search + admissible heuristic
- Searching with consistent heuristics

Derivation of heuristics

Goals for the lecture

- Analysis of different heuristics
- Tools for analysis
- Relaxation as general method to obtain heuristics
- Importance of efficient implementation

Analysis of Manhattan-distance heuristic

Manhattan-distance heuristic is:

- **Safe** (trivial because $h(s) < \infty$ for all states)
- **Goal-aware** (because every tile at goal has distance $= 0$)
- **Admissible** (because of ?)
- **Consistent** (because of ?)

Analysis of straight-line distance heuristic

Straight-line distance heuristic is:

- **Safe** (trivial because $h(s) < \infty$ for all states)
- **Goal-aware** (because sl distance of goal is 0)
- **Admissible** (shortest path between two points is straight line)
- **Consistent** (euclidean distances satisfy triangular inequality)

Analysis of MST heuristic

MST heuristic for TSP is:

- **Safe** (trivial because $h(s) < \infty$ for all states)
- **Goal-aware** (because when tour is complete MST is empty)
- **Admissible** (because of ?)
- **Consistent** (because of ?)

Constructing heuristics: relaxations

A good heuristic should:

- Provide good estimates (also known as to be informative)
- Be efficiently computable

Standard method to obtain heuristic functions is:

- Relax or simplify problem to make it easier
- Solve relaxed problem and use cost of solution as heuristic value

Relaxation in 15-puzzle: Manhattan distance

In 15-puzzle:

- Movements swap the blank with an adjacent tile
- **Implied constraint:** no two or more tiles can be at same position
- **Relax constraint** so that tiles can move into adjacent positions even if non empty

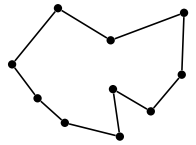
Consequences:

- A tile can be moved into its goal location in a number of movements equal to its Manhattan distance
- Manhattan distance of a state is **minimum number** of movements required to solve the configuration in **relaxed problem**

Relaxations in TSP: MST and AP heuristics

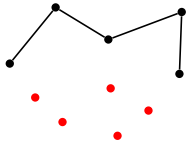
A tour for n cities is a subgraph such that:

- It is connected
- The degree of each vertex is 2

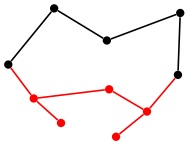


Given node in search tree (i.e. partial tour):

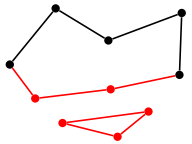
- Optimal cost when second condition relaxed away = **MST heuristic**
- Optimal cost when first condition relaxed away = **AP heuristic**



partial tour



MST heuristic



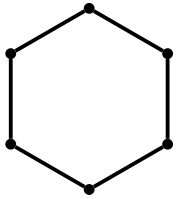
AP heuristic

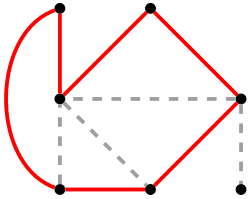
Graph embeddings

A tool for analysis/design of heuristics is idea of **graph embedding**

Intuitively, graph G is embedded in H if it can be "mapped" inside H

G





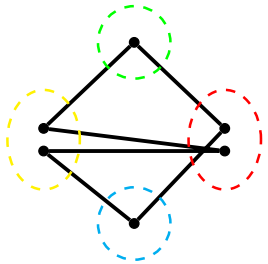
H_1

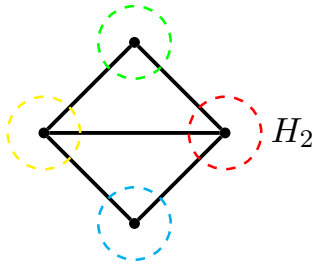
Graph embeddings

A tool for analysis/design of heuristics is idea of **graph embedding**

Intuitively, graph G is embedded in H if it can be "mapped" inside H

G



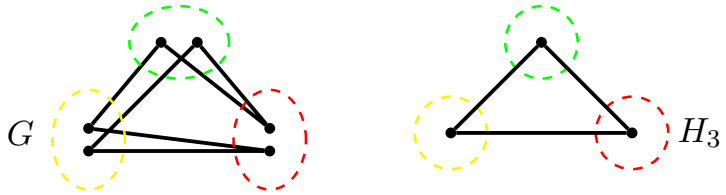


H_2

Graph embeddings

A tool for analysis/design of heuristics is idea of **graph embedding**

Intuitively, graph G is embedded in H if it can be "mapped" inside H



Graph embeddings

A tool for analysis/design of heuristics is idea of **graph embedding**

Intuitively, graph G is embedded in H if it can be "mapped" inside H

Graph embedding: formal definition

Let $G = \langle V_G, E_G, c_G \rangle$ and $H = \langle V_H, E_H, c_H \rangle$ be two graphs with edge costs (they can be multi-graph, directed or undirected)

G is **embedded** in H if there is (embedding) function $\alpha : V_G \to V_H$ that maps states of G into states of H such that:

- $(u, v) \in E_G \implies (\alpha(u), \alpha(v)) \in E_H$
- $c_H(\alpha(u), \alpha(v)) \leq c_G(u, v)$ for all edges $(u, v) \in E_G$

If so, we say that α **embeds** G **in** H and write $\alpha : G \leq H$

If $\alpha : G \leq H$, paths π in G are transformed into paths $\alpha(\pi)$ in H of equal or lesser cost; i.e. $c_H(\alpha(\pi)) \leq c_G(\pi)$

Heuristics produced from graph embeddings

Let's consider:

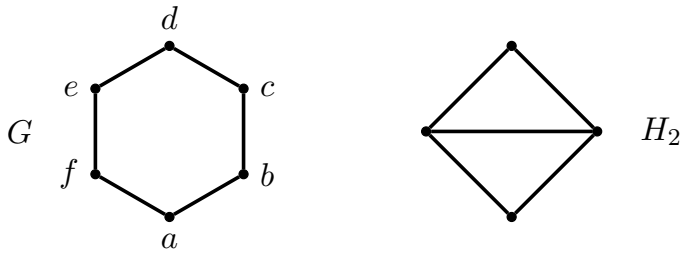
- State model $P = \langle S, A, s_{init}, S_G, f, c \rangle$ and induced labeled graph $G(P) = \langle V, L, E \rangle$ with edge costs c
- A graph $H = \langle V_H, E_H, c_H \rangle$
- Embedding $\alpha : \langle G(P), c \rangle \leq H$
- Vertices $V^\star \subseteq V_H$ such that $V^\star \supseteq \{\alpha(s) : s \in S_G\}$

We can use H and f to define **heuristic functions**:

- $h_\Pi(s) \doteq c_H(\Pi(s))$ where Π is an **algorithm** that on input s outputs a path in H from $f(s)$ to a state in $V^\star$
- $h_\alpha(s) \doteq \delta_{c_H}(\alpha(s), V^\star)$ where $\delta(x, Y) \doteq \min_{y \in Y} \delta(x, y)$

For any H and $\alpha : G \leq H$, the heuristic h_α is admissible and consistent

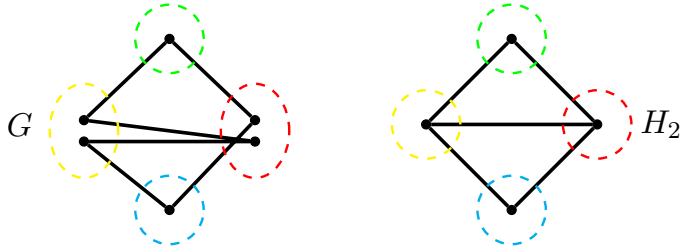
Example of heuristics obtained from embeddings



dist.	a	b	c	d	e	f
a	0	1	2	3	2	1

original distances

Example of heuristics obtained from embeddings

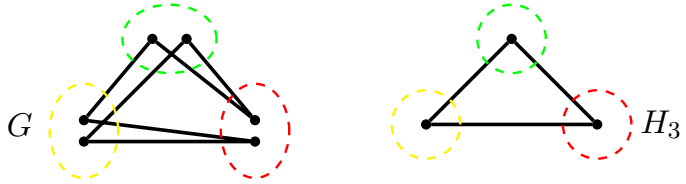


dist.	a	b	c	d	e	f
a	0	1	2	3	2	1
a	0	1	2	1	1	1

original distances

$\alpha_2 : G \leq H_2$

Example of heuristics obtained from embeddings



dist.	a	b	c	d	e	f
a	0	1	2	3	2	1
a	0	1	2	1	1	1
a	0	1	0	1	1	1

original distances

$\alpha_2 : G \leq H_2$

$\alpha_3 : G \leq H_3$

Example of heuristics obtained from embeddings

dist.	a	b	c	d	e	f
a	0	1	2	3	2	1
a	0	1	2	1	1	1
a	0	1	0	1	1	1
a	0	1	0	1	0	1

original distances

$\alpha_2 : G \leq H_2$

$\alpha_3 : G \leq H_3$

$\alpha_4 : G \leq H_4$

Manhattan-distance heuristic

Manhattan-distance is a graph embedding relaxation:

- Graph G is space of configuration for 15-puzzle
- Graph H is G augmented with configurations in which more than one tile can be at the same location, and also with edges
- Edge costs in H are 1
- Embedding $\alpha : G \leq H$ is the **identity function**
- Subset $V^\star \subseteq V_H$ is singleton with goal configuration in 15-puzzle

Manhattan distance is thus **admissible** and **consistent**

Abstractions

An abstraction of a search problem P is an embedding of the induced graph $G(P)$ into another graph H

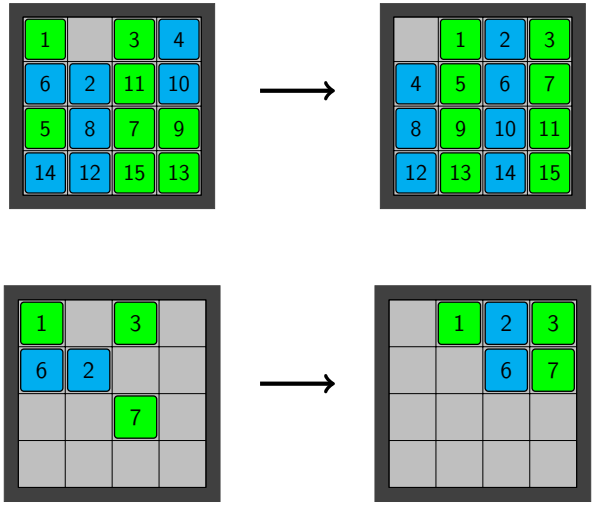
Abstraction are constructed by **identifying** states in P and making them **equivalent** in H

In PSVN, abstractions are constructed in two ways:

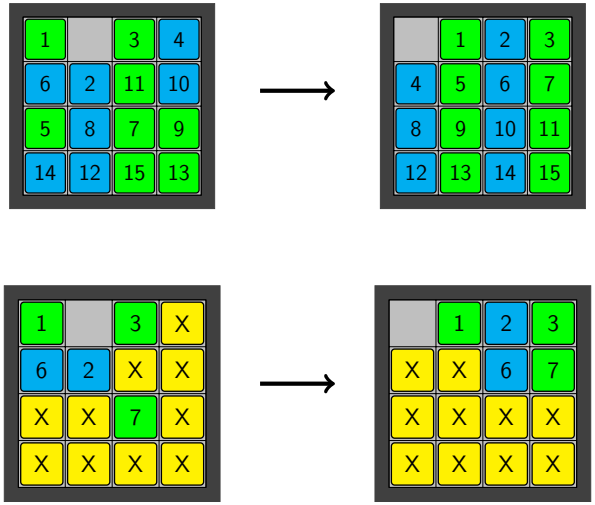
- projecting (removing) vector positions
- reducing the domain of selected vector positions

Ideally, both ways result in a smaller abstract problem, yet projections sometimes generate bigger problems

Abstractions: 15-puzzle (type 1)



Abstractions: 15-puzzle (type 2)



Combining abstractions

Let $\alpha_1 : G \leq H_1, \ldots, \alpha_k : G \leq H_k$ be k abstractions for problem P with induced graph G

We know that the heuristic function h_{α_i} is **admissible** and **consistent**

The heuristic $h = \max_i h_{\alpha_i}$ defined as

$$h_{\max}(s) \;=\; \max_i h_{\alpha_i}(s) \;=\; \max_i \delta_{c_{H_i}}(\alpha_i(s), V^\star_{H_i})$$

is also **admissible** and **consistent**

Can we obtain a better combination of values?

Additive abstractions

We say that the abstractions $\alpha_1 : G \leq H_1, \ldots, \alpha_k : G \leq H_k$ are **additive** if the following heuristic is **admissible**:

$$h_\Sigma(s) \;=\; \sum_i h_{\alpha_i}(s) \;=\; \sum_i \delta_{c_{H_i}}(\alpha_i(s), V^\star_{H_i})$$

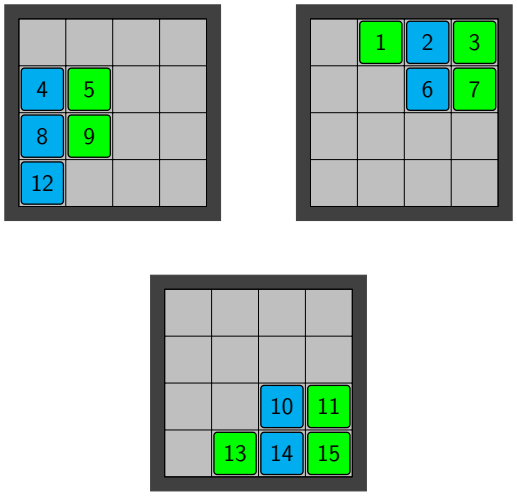
A **sufficient condition** for the additivity of the abstractions is that for each state s and action $a \in A(s)$:

$$\alpha_i(f(s,a)) \;\neq\; s \qquad \text{AND} \qquad c_{H_i}(\alpha_i(s), \alpha_i(f(s,a))) \;>\; 0$$

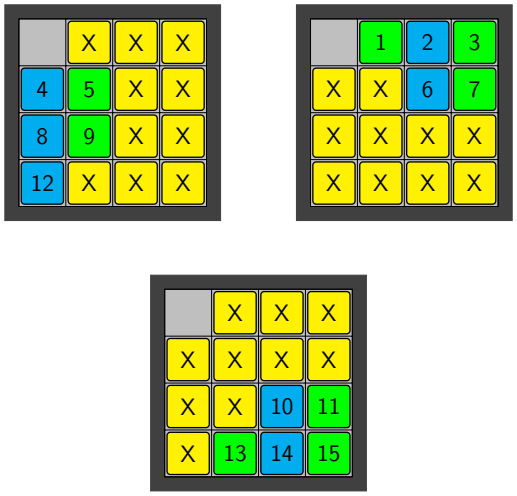
for **at most one abstraction** i (f is the transition function in P)

In such case, h_Σ is also **consistent**

Additive abstractions: 15-puzzle (type 1)



Additive abstractions: 15-puzzle (type 2)



Additive iff cost of moving yellow tiles is zero in all abstractions

Pattern databases

A pattern database is the solution of an abstraction **stored in memory** in an efficient manner

In order to construct the pattern database, the size of the abstraction should be small enough to fit the available amount of RAM

The abstraction can be stored in a **hash table** or in an array if a **perfect hash function** is available

Once the abstraction is solved and stored in memory, heuristic computations take $O(1)$ time

Solving the abstraction may be expensive but the cost can be **amortized** over many instances

Efficient implementation of heuristics

Importance of efficient implementations

instance	#generated
01	18,910,192
02	388,009,351
05	153,904,617
06	11,433,741
18	2,459,171,888
33	619,583,956
59	1,413,514,386
60	3,250,487,394
82	4,863,382,216
88	6,329,954,135
100	38,527,126
total	37,336,890,306

IDA* with Manhattan distance on Korf's 100 instances for 15-puzzle

(Note: nodes generated in order: up, right, down, left)

Manhattan-distance heuristic: computation

```
1  struct State15Puzzle : AbstractState {
2      char pos[16]  % pos[i] contains the 'tile' in ith-position
3      char blank    % contains position of blank
4  }
5
6  unsigned manhattanI(State15Puzzle puzzle) {
7    unsigned h = 0
8    for i = 0 to 15
9        if puzzle.pos[i] != 0
10           h += abs(xcoord(i) - xcoord(puzzle.pos[i]))
11           h += abs(ycoord(i) - ycoord(puzzle.pos[i]))
12   return h
13 }
```

- `xcoord()` and `ycoord()` compute x- and y-coordinates of index
- $O(n^2)$-time implementation of Manhattan for (n^2-1)-puzzle
- Can we do better?

Manhattan-distance heuristic: computation

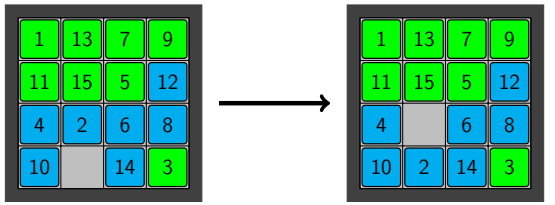
```
1  unsigned mtable0[16] = { 0, 0, 0, 0, ... 0, 0, 0, 0 }
2  unsigned mtable1[16] = { 1, 0, 1, 2, ... 4, 3, 4, 5 }
3  ...
4  unsigned mtable15[16] = { 6, 5, 4, 3, ... 3, 2, 1, 0 }
5  unsigned[] mtable[16] = { mtable0, mtable1, ... mtable15 }
6
7  unsigned manhattanII(State15Puzzle puzzle) {
8    unsigned h = 0
9    for i = 0 to 15
10       h += mtable[puzzle.pos[i]][i]
11   return h
12 }
```

- $O(n^2)$-time implementation of Manhattan for (n^2-1)-puzzle (this one has a better hidden constant)
- Trade space for efficiency **(recurring technique)**
- Can we do better?

Manhattan-distance heuristic: computation

Exploit incremental computation **(recurring technique)**:

- Given parent's heuristic value, compute new heuristic value
- A movement changes the Manhattan distance of just one tile
- It's enough to adjust heuristic value for such tile



Manhattan distance for tile 2 changes by $\Delta h = +1$

Manhattan-distance heuristic: computation

- Given heuristic value of parent, computes heuristic value of node
- $O(1)$-time implementation of Manhattan for (n^2-1)-puzzle
- Exploits incremental computation

```
1  unsigned manhattanIII(State15Puzzle puzzle, Node parent) {
2    unsigned tile = puzzle.pos[parent.state.blank]
3    unsigned oldpos = parent.state.blank
4    unsigned newpos = puzzle.blank
5    return parent.state.h-value + delta[tile][oldpos][newpos]
6  }
7
8  unsigned d0[16][16] = { X, X, X, X, ... X, X, X, X }  % not used
9  unsigned d1[16][16] = { 0,-1, X, X,   1, X, ...  X,   % opos = 0
10                         1, 0, 1, X,   X, 1, ...  X,   % opos = 1
11                         ...                           % ...
12                         X, X, X, X, ... X, X, 1, 0 }  % opos = 15
13 ...
14 unsigned[][] delta[16] = { d0, d1, ... d15 }
```


Manhattan-distance heuristic: comparison

instance	#generated	Manhattan-distance heuristic I	II	III
01	18,910,192	1.04	0.44	0.34
02	388,009,351	21.28	9.05	7.10
05	153,904,617	8.46	3.61	2.81
06	11,433,741	0.63	0.26	0.20
18	2,459,171,888	133.43	57.05	45.42
33	619,583,956	33.72	14.35	11.45
59	1,413,514,386	77.19	33.26	27.35
60	3,250,487,394	175.78	75.58	61.19
82	4,863,382,216	264.73	113.27	89.10
88	**6,329,954,135**	**343.17**	**147.69**	**116.11**
100	38,527,126	2.09	0.91	0.69
total	37,336,890,306	2,034.03	875.08	751.98

Intel Core i5 3.4GHz

IDA* on Korf's 100 instances of 15-puzzle (time in seconds)

Summary

- Relaxation as general method to obtain heuristics
- Graph embeddings as a tool for analysing heuristics
- Manhattan distance, MST and AP are admissible and consistent
- Abstractions and pattern databases
- Efficient implementation is crucial

Iterative deepening A* (IDA*)

Goals for the lecture

– Motivation for IDA*

– IDA* and its properties

– Success stories for IDA*

– Difficulties of IDA*

Motivation for IDA*

Like with uniform-cost search, **A* consumes too much memory**

Challenge is to develop an **optimal linear-space algorithm** able to use the information provided by the heuristic

Solution is Richard Korf's iterative deepening A* (IDA*) [Korf, 1985]

IDA*

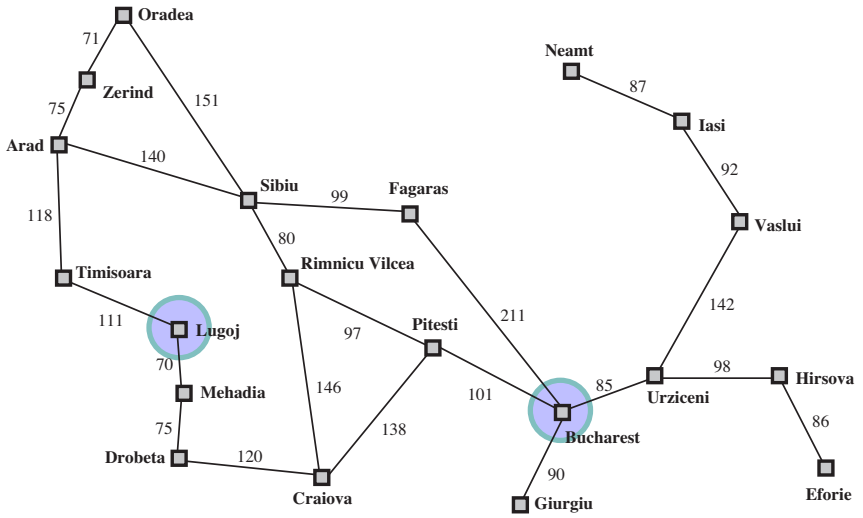
In iterative deepening depth-first search:

- The search tree is **sliced into subtrees of increasing depth**
- Each subtree comprises the nodes whose depth is no bigger than current depth bound
- The depth bound increases by 1 after each unsuccessful search

In IDA*:

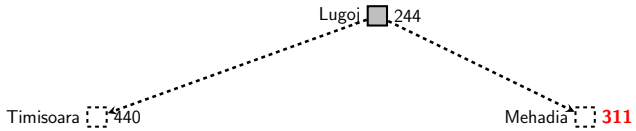
- The search tree is **sliced using f-values**
- Each subtree comprises the branches whose f-values are no bigger than the current f-bound
- The f-bound increases as the minimum f-value of the nodes generated but not visited in last search

Example: traveling in Romania [AIMA]



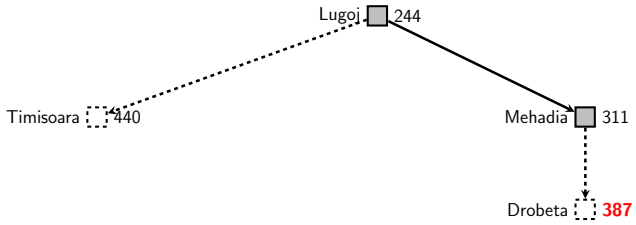
Task: find optimal path from **Lugoj** to **Bucharest**

IDA*: example of subtrees



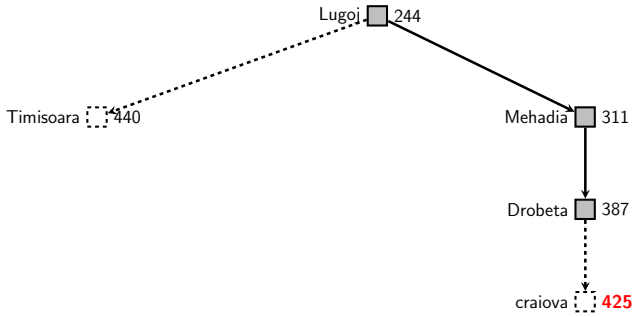
Bounded sub-tree: bound on f-cost is $h(s_0) = 244$, next bound is 311

IDA*: example of subtrees



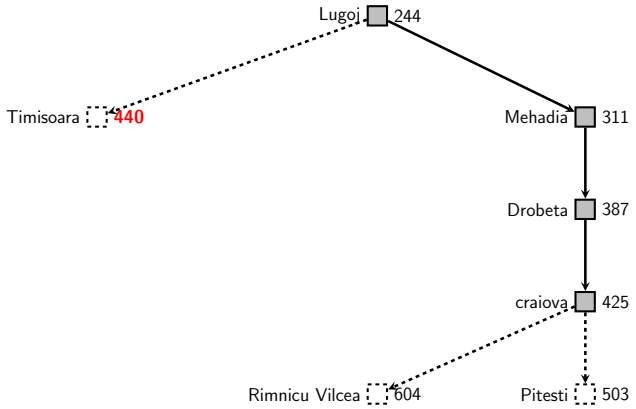
Bounded sub-tree: bound on f-cost is 311, next bound is 387

IDA*: example of subtrees



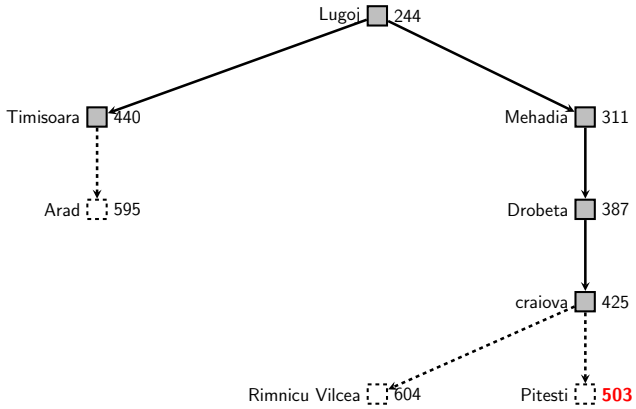
Bounded sub-tree: bound on f-cost is 387, next bound is 425

IDA*: example of subtrees



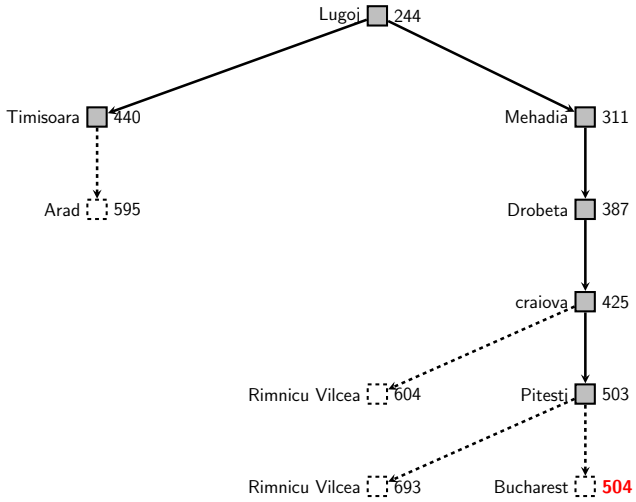
Bounded sub-tree: bound on f-cost is 425, next bound is 440

IDA*: example of subtrees



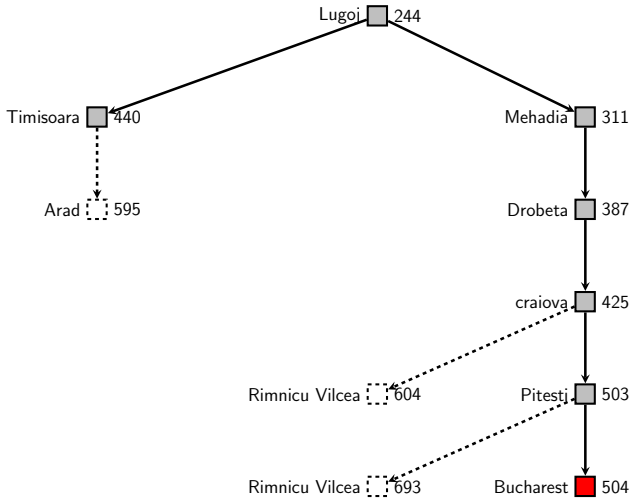
Bounded sub-tree: bound on f-cost is 440, next bound is 503

IDA*: example of subtrees



Bounded sub-tree: bound on f-cost is 503, next bound is 504

IDA*: example of subtrees



Bounded sub-tree: bound on f-cost is 504

IDA*: pseudocode

```
1  Node ida-search():
2      Node root = make-root-node(init())
3      unsigned bound = h(init())
4
5      % perform f-limited searches with increasing cost bounds
6      while true
7          pair<Node,unsigned> p = f-bounded-dfs-visit(root, bound)
8          if p.first != null return p.first
9          bound = p.second
10
11 pair<Node,unsigned> f-bounded-dfs-visit(Node n, unsigned bound):
12     % base cases
13     f = n.g + h(n.state)
14     if f > bound return (null,f)
15     if n.state.is-goal() return (n,n.g)
16
17     t = ∞
18     foreach <s,a> in n.state.successors()
19         Node n' = n.make-node(s, a)
20         pair<Node,unsigned> p = f-bounded-dfs-visit(n', bound)
21         if p.first != null return p
22         t = min(t,p.second)
23     return (null,t)    % failure: there is no path from node n to goal
```


Properties of IDA*

- **Complete:** complete if goal is reachable and positive costs (incomplete otherwise)
- **Optimality:** it returns a **minimum-cost** path
- **Time complexity:** $O(b^d) = O(b^{c^*/c_{\min}})$
- **Space complexity:** $O(bd) = O(b\,c^*/c_{\min})$

Time and space complexities calculated in **canonical search tree** with branching factor b and $c_{\min} > 0$ where **minimum-cost** goal appears at depth d with cost c^*

Time complexity assumes number of nodes generated but not visited at each iteration form a **geometric series with rate** b

Impact of IDA*

- 15-puzzle and 24-puzzle
- $3 \times 3 \times 3$ Rubik's cube
- Top Spin
- Pancake problem

IDA* is not good on:

- TSP
- Automated planning

Difficult problems for IDA*

Assumptions for success of IDA*:

- number of depth-first searches is (relatively) small (e.g. linear in depth)
- goals may appear at any depth
- number of duplicates is not too much

If some of these assumptions is violated, IDA* may not be effective:

- each search visits only one new node: large number of iterations
- all goals appear at same depth: IDA* wastes a lot of time
- too many duplicates: makes sense to use memory to remove them

Comparing A* with IDA*

Even though IDA* explores duplicates that A* avoids, IDA* may be a faster algorithm than A* in some problems

It all depends in the relation between the **average time to generate nodes** and the **proportion of duplicates visited by IDA***

Formally, let $t_{\text{A*}}$ and $t_{\text{IDA*}}$ be the average time for node generation for A* and IDA* respectively on a given problem (clearly, $t_{\text{IDA*}} \leq t_{\text{A*}}$)

Also, let $n_{\text{A*}}$ and $n_{\text{IDA*}}$ be the number of nodes generated by A* and IDA* respectively (clearly, $n_{\text{A*}} \leq n_{\text{IDA*}}$)

The time taken by A* and IDA* is approximately $n_{\text{A*}} \times t_{\text{A*}}$ and $n_{\text{IDA*}} \times t_{\text{IDA*}}$ respectively

Comparing A* with IDA*

Therefore, IDA* is faster than A* on given problem when:

$$n_{\text{IDA*}} \times t_{\text{IDA*}} < n_{\text{A*}} \times t_{\text{A*}} \quad \Longrightarrow \quad \frac{n_{\text{IDA*}}}{n_{\text{A*}}} < \frac{t_{\text{A*}}}{t_{\text{IDA*}}}$$

That is, when **fraction of duplicates** in the search tree for IDA* is less than the **speed up in node generation** achieved by IDA*

We conclude that it is very important to:

- improve node generation
- improve heuristic calculation (impact on node generation)
- prune as many duplicates as possible while affecting node generation as little as possible

Efficient implementation of IDA*

In some cases it is possible to make a more efficient implementation of IDA*. We assume that

- all operators are invertible; i.e. each action a has an inverse a^{-1} such that $s = f(f(s, a), a^{-1})$ for all s and $a \in A(s)$
- the heuristic h is safe; i.e. if $h(s) = \infty$, the goal is unreachable
- $h(s) = 0$ iff s is a goal state

These conditions are met by large number of problems and heuristics

Under these conditions, we can **exploit the recursion stack** to:

- get rid of the node data structure
- use a single global state data structure

Efficient implementation of IDA*: pseudocode

```
global State state
global vector<Action> path

void ida-search():
    state = init()
    unsigned bound = h(init())
    % perform f-limited searches with increasing cost bounds
    while true
        pair<bool,unsigned> p = f-bounded-dfs-visit(bound, 0)
        if p.first return
        bound = p.second

pair<bool,unsigned> f-bounded-dfs-visit(unsigned bound, unsigned g):
    % base cases
    h = h(state)
    f = g + h
    if f > bound return (false,f)
    if h == 0 return (true,g)    % h(s) = 0 iff s is goal

    t = ∞
    foreach action a that is applicable at state
        cost = g + c(state, a)
        state = f(state, a)
        if h(state) < ∞
            path.push-back(a)
            pair<bool,unsigned> p = f-bounded-dfs-visit(bound, cost)
            if p.first return p
            t = min(t,p.second)
            path.pop-back(a)
        state = f(state, inverse(a))
    return (false,t)    % failure: there is no path from node n to goal
```


Summary

- IDA* is an optimal linear-space algorithm that exploits heuristic information
- IDA* is for A* what iterative deepening depth-first search is for breadth-first search
- IDA* had been quite successful in a number of applications
- In certain problems, IDA* is not the best algorithm

Hill climbing

Goals for the lecture

- Learn about hill climbing
- Learn about enforced hill climbing

What to do with the heuristic information?

Heuristic functions provide **guidance information** to reach the goal

Three approaches for using information:

- **Consider heuristic as perfect estimate (hill climbing)**
- Use it to order nodes for expansion (greedy best-first search)
- Combine with information gathered during search (best-first search)

Hill climbing

Hill climbing is a **local search** algorithm

- Starting at the initial state,
- hill climbing always moves to a **best** child
  (the one closest to the goal as measured by the heuristic)

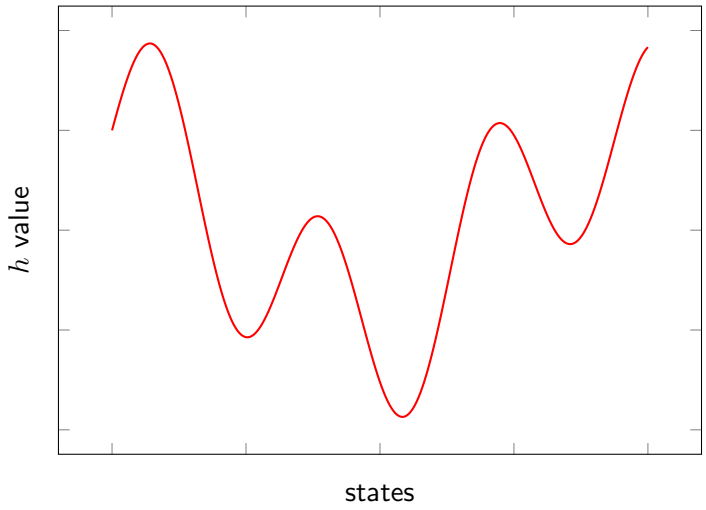
Hill climbing: pseudocode

```
1  % Hill climbing
2  Node hill-climbing(Heuristic h):
3      Node n = make-root-node(init())
4
5      while true
6          if n.state.is-goal() return n
7
8          % expansion
9          Succ = ∅
10         foreach <a,s> in n.state.successors()
11             if h(s) != ∞
12                 Succ = Succ ∪ { n.make-node(s, a) }
13         if Succ == ∅ break
14
15         % greedy selection of best child
16         n = select node from Succ minimizing h (random tie breaking)
17
18     return null
```


Properties of hill climbing

- **Completeness:** incomplete, it can loop forever after getting trapped in a **local minimum**
- **Optimality:** suboptimal
- **Time complexity:** it can loop forever
- **Space complexity:** constant space (more accurate: $O(b)$)

Local minima



(children of state x are the states to left and right of x)

Remarks on hill climbing

- Hill climbing is different from greedy best-first search
- GBFS maintains a queue and select for expansion the node in queue with minimum heuristic value
- With duplicate elimination, GBFS is complete
- Variations on hill climbing include:
  - different tie-breaking strategies
  - random restarts
  - different mechanisms for escaping local minima

Enforced hill climbing

Variation on hill climbing that makes it into a **complete algorithm**

The idea is to **wrap** hill climbing with a **breadth-first search**

Given a node n, we use breadth-first search to look for a descendant n' of n with a heuristic value (n') better than $h(n)$

Enforced hill climbing is an algorithm that is widely used when you are satisfied with any solution and there is a reasonable heuristic

Enforced hill climbing: pseudocode

```
1  % Breadth-first search to improve h-value of node n0
2  Node improve(Node n0, Heuristic h):
3      Queue q                    % FIFO queue
4      q.insert(n0)
5      set-color(n0.state, Gray)  % Use a fresh hash table
6
7      while !q.empty()
8          Node n = q.pop()
9          if get-color(n.state) != Black
10             if h(n.state) < h(n0.state) return n
11
12             % expansion
13             foreach <a,s> in n.state.successors()
14                 q.insert(n.make-node(s, a))
15             set-color(n.state,Black)
16     return null                % failure: node n0 can't be improved
17
18 % Enforced hill climbing
19 Node enforced-hill-climbing(Heuristic h)
20     Node n = make-root-node(init())
21     while n != null && !is-goal(n.state)
22         n = improve(n, h)
23     return n
```

Properties of enforced hill climbing

- **Completeness:** if **connected space** (e.g. undirected graphs) and $h(s) = 0$ for all and only goal states
- **Optimality:** suboptimal
- **Time complexity:** $O(S)$
- **Space complexity:** $O(S)$

Summary

- Hill climbing
  - Considers heuristic 100% accurate
  - Incomplete and suboptimal
  - Very fast and constant memory
- Enforced hill climbing
  - Combines hill climbing with breadth-first search
  - Complete but suboptimal
  - Often used in hard combinatorial problems

Weighted A* and IDA*

Goals for the lecture

- How to trade quality of solution (optimality) for performance
- Develop algorithms that run faster but return suboptimal solutions (but that still offer some guarantees)
- Weighted A* and IDA* (and others as well)

Empirical observation

Algorithms that use non-admissible heuristics:

– typically run faster than when using admissible heuristics

– they can't provide any guarantee on the quality of the solution

IDEA: given admissible heuristic h, make h non-admissible with the aim of improving performance of search

Making the heuristic non-admissible

Give admissible heuristic h, want to **"tweak"** h into h' so that

– h' retains as much as possible the **"information"** in h

– h' is non-admissible

Different ways to do this:

– ~~Addition of constant: $h'(s) = h(s) + \beta$~~ (h' isn't goal aware)

– Multiplication by constant: $h'(s) = \alpha h(s)$

– ~~Affine transformation: $h'(s) = \alpha h(s) + \beta$~~ ($h'$ isn't goal aware)

– ~~Non-linear mappings; e.g. $h'(s) = \log h(s)$~~ (don't know)

Multiplication by constant factor: weightening

Given admissible heuristic h

Use evaluation function $f(n) = g(n) + (1 + \epsilon) \times h(n)$ with fixed $\epsilon > 0$

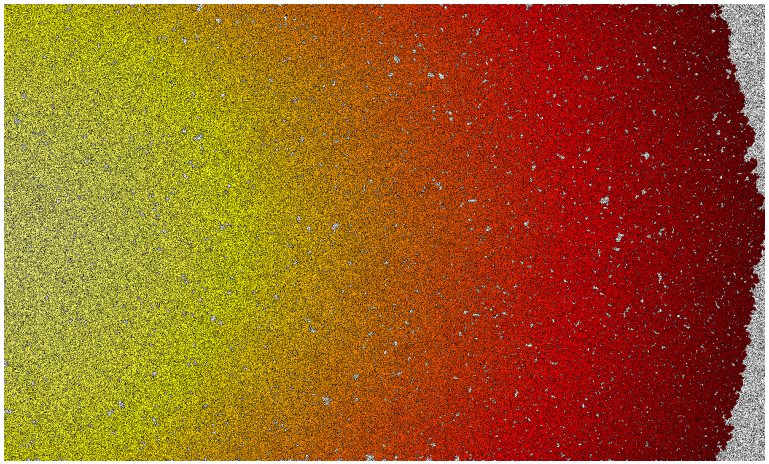
It doesn't make sense to multiply by constant $\alpha < 1$

Weightening can be used in all informed algorithms seen so far

Example: impact of weightening on performance

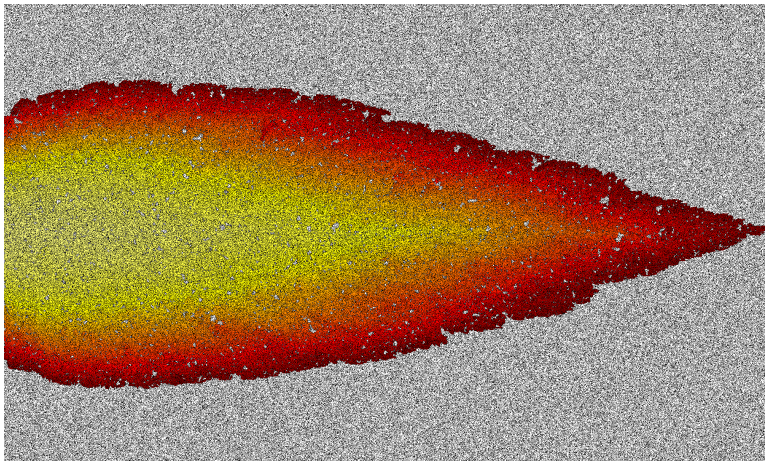
- Problem is a path-finding task in a 4-connected grid
- Heuristic is Manhattan distance assuming no obstacles
- Initial cell at the middle row and leftmost column
- Goal cell is at the middle row and rightmost column
- Colors for cells:
  - black cells are obstacles (i.e. non-traversable cells)
  - white cells were not expanded in the search
  - yellow cells were expanded early in the search
  - dark red cells were expanded late in the search

Example: uniform-cost search



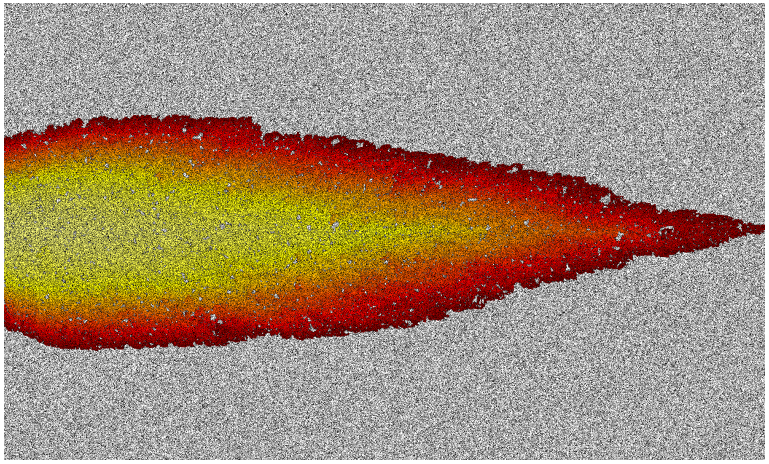
[Image by Jordan Thayer. Http://www.jordanthayer.com]

Example: A*



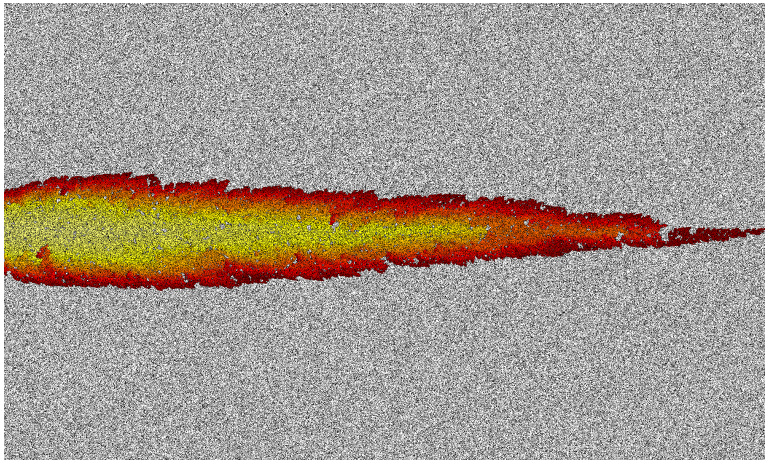
[Image by Jordan Thayer. Http://www.jordanthayer.com]

Example: $(1+\epsilon)$-weighted A* with $\epsilon = 1.1$



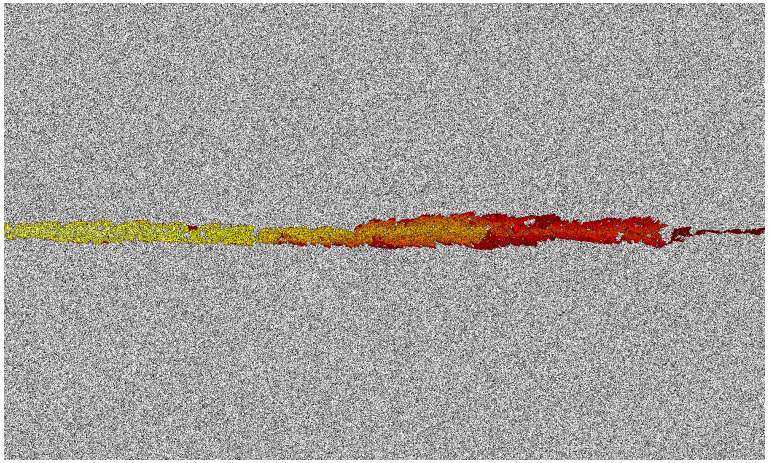
[Image by Jordan Thayer. Http://www.jordanthayer.com]

Example: $(1+\epsilon)$-weighted A* with $\epsilon = 1.3$



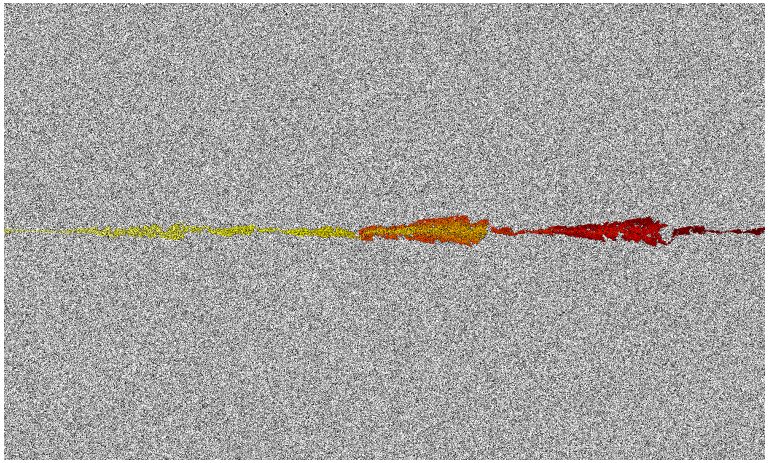
[Image by Jordan Thayer. Http://www.jordanthayer.com]

Example: $(1+\epsilon)$-weighted A* with $\epsilon = 1.5$



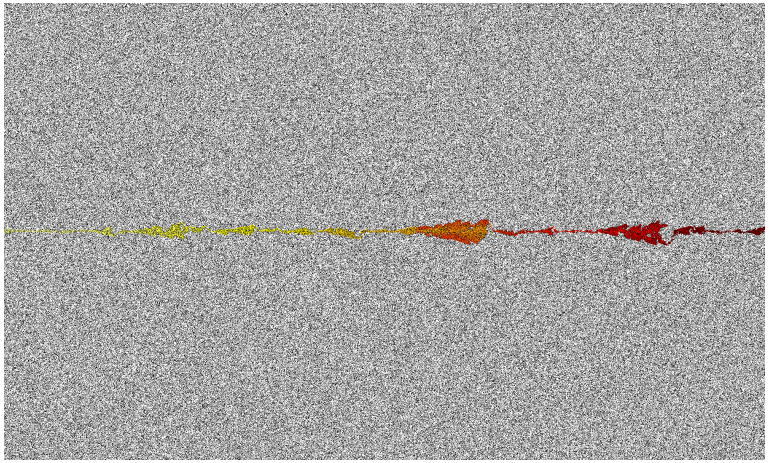
[Image by Jordan Thayer. Http://www.jordanthayer.com]

Example: $(1+\epsilon)$-weighted A* with $\epsilon = 1.7$



[Image by Jordan Thayer. Http://www.jordanthayer.com]

Example: $(1+\epsilon)$-weighted A* with $\epsilon = 2$



[Image by Jordan Thayer. Http://www.jordanthayer.com]

Theoretical guarantees

Let h be an admissible heuristic and $h' = (1+\epsilon)\, h$ for $\epsilon \geq 0$

A* and IDA* (and other optimal and informed algorithms) return a solution π that is $(1+\epsilon)$-**optimal**:

$$c(\pi) \;\leq\; (1+\epsilon)\, c^*$$

where c^* is the cost of an optimal solution

Limit cases

Evaluation function: $f(n) = g(n) + \alpha h(n)$

α	reduces to	optimality factor
0	uniform-cost search	1 (optimal)
1	A*	1 (optimal)
$1+\epsilon$	weighted A*	$1+\epsilon$
∞	greedy best-first search	unbounded

Experimental results

IDA* with evaluation function $f(n) = g(n) + \lfloor w\, h(s) \rfloor$

weight	#generated	average length	average time	average suboptimality	average speedup
$w = 1.0$	37,336,890,306	53.05	7.519	—	—
$w = 1.1$	20,484,854,270	53.19	4.165	1.002	2.2
$w = 1.2$	7,046,484,202	53.61	1.459	1.010	7.0
$w = 1.3$	2,422,127,610	54.57	0.504	1.028	24.8
$w = 1.4$	1,131,895,475	55.29	0.237	1.042	84.0
$w = 1.5$	487,832,741	56.39	0.102	1.063	279.7
$w = 1.6$	267,106,362	58.23	0.056	1.097	359.7
$w = 1.7$	107,159,144	60.41	0.023	1.138	761.0
$w = 1.8$	42,440,789	62.71	0.009	1.181	1,617.6
$w = 1.9$	36,310,749	65.07	0.008	1.225	1,560.4
$w = 2.0$	26,790,886	67.99	0.006	1.280	1,779.9
$w = 2.5$	15,553,801	82.77	0.003	1.558	4,080.8
$w = 3.0$	8,379,728	98.25	0.002	1.849	4,063.1
$w = 5.0$	9,978,522	160.43	0.002	3.018	6,039.5

Intel Core i5 3.4GHz

Weighted IDA* on Korf's 100 instances of 15-puzzle (time in seconds)

Guarantees on performance

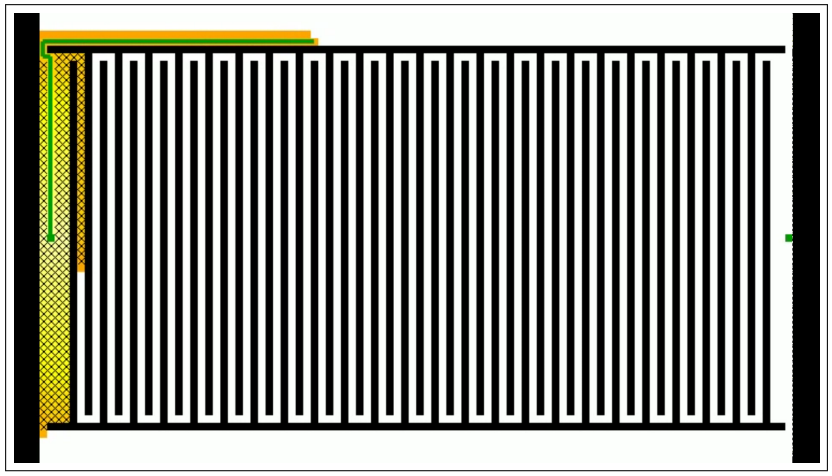
Does weighted algorithms offer any guarantee on performance?

No, there are no guarantees whatsoever

Empirically, the performance of weighted algorithms increase substantially with the weight

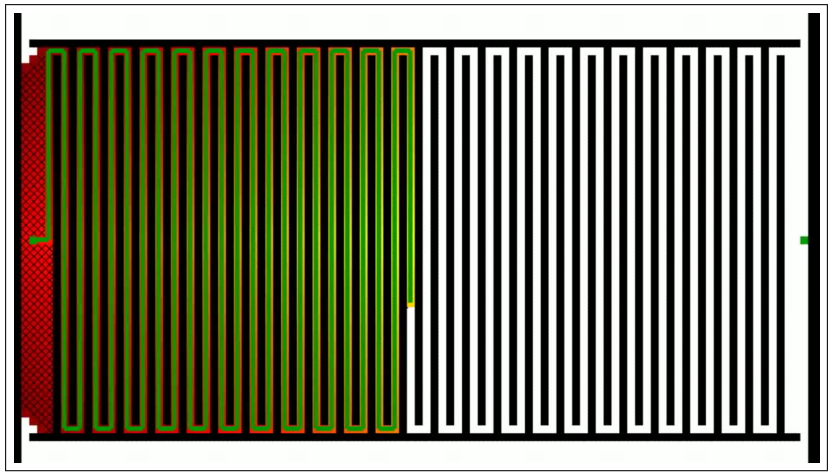
However, there are known **pathological cases** in which a weight can harm performance

A* on a ladder problem (video)



[Video by Jordan Thayer. `Http://www.jordanthayer.com`]

Weighted A* on a ladder problem (video)



[Video by Jordan Thayer. Http://www.jordanthayer.com]

Summary

- Heuristic function can be multiplied by factor $(1 + \epsilon)$ to improve performance while controlling loss in quality
- WA* and WIDA* are well-known and heavily used algorithms
- Basic idea can be further exploited to develop algorithms that:
  - compute a solution fast
  - keep improving solution while time is available
  - if given enough time, solution becomes optimal

  This type of algorithm is called an **anytime algorithm**

Branch and bound

Goals for the lecture

- Depth-first branch and bound algorithm
- Parallelisation and super-linear speed ups
- General branch and bound

Depth-first branch and bound

DFBnB performs a depth-first traversal of the search tree using **linear memory** to find a "best solution"

A solution can be either a path in a graph or, more generally, a configuration in a combinatorial space

We first focus on finding optimal paths and then move to the more general framework

We assume:

- positive action costs; i.e. $c(s,a) > 0$ for all $s \in S$ and $a \in A(s)$
- an admissible heuristic function $h : S \to \mathbb{R}^{\geq 0}$

Depth-first branch and bound: pruning

The idea is to maintain **global variables** that store the best current solution and to prune branches of the tree that **provably** cannot improve the best current solution

A node n corresponding to a path π in the search tree "represents" all paths with prefix π. Let's denote the set of all such paths as $\Pi(n)$

Since the heuristic is admissible, the value $f(n)$ is a **lower bound** on the cost of all paths in $\Pi(n)$

Hence, if the cost of the best current solution is α, then no path in $\Pi(n)$ can improve the best current solution when

$$f(n) = g(n) + h(n.state) \;\geq\; \alpha$$

Pruning rule in depth-first branch and bound!

Depth-first branch and bound: pseudocode

```
1  global unsigned best-bound = ∞
2  global Node best-solution = null
3
4  % Depth-first branch and bound
5  Node depth-first-branch-and-bound():
6      Initialize best-bound and best-solution with some solution
7      Node root = make-root-node(init())
8      depth-first-bnb-visit(root)
9      return best-solution
10
11 % Depth-first traversal of search tree
12 void depth-first-bnb-visit(Node n):
13     % test if branch can be pruned
14     if n.g + h(n.state) >= best-bound return    % prune node
15
16     % test whether we found a better solution
17     if n.state.is-goal()
18         best-bound = n.g
19         best-solution = n
20         return
21
22     % depth-first recursion
23     foreach <s,a> in n.state.successors()
24         depth-first-bnb-visit(n.make-node(s,a))
```


Properties of depth-first branch and bound

- **Complete:** it is a complete algorithm (assuming safeness, positive costs, and solution), it may get trapped in infinite branch if there is no solution
- **Optimality:** yes if heuristic is admissible
- **Time complexity:** $O(b^d)$
- **Space complexity:** $O(bd)$

If the problem has zero costs, the depth-first traversal can get trapped in an infinite branch of zero cost

Time and space complexities calculated in **canonical search tree** with branching factor b, height d, and unit costs

Performance of depth-first branch and bound

Two **important events** in any run of branch and bound:

- Optimal solution is found (elapsed time to find optimal solution)
- Algorithm terminates (elapsed time from previous event to termination)

The first time measures difficulty of **finding an optimal solution**

The second time measures difficulty of **proving optimality**

In general, the second time is much larger than the first time (i.e. proving optimality is more difficult than finding an optimal solution)

Depth-first branch and bound vs. IDA*

DFBnB and IDA* are both depth-first traversals of the search tree that look for an optimal solution

Which one is better?

In problems where solutions appear at different depths (15-puzzle), **DFBnB is typically the wrong algorithm** because

- DFBnB may go beyond the goal depth and spend a lot of time exploring a huge subtree in which no optimal solution exists

On the other hand, when all solutions appear at the same depth (like TSP), **IDA* is the wrong algorithm** because

- IDA* performs traversals on subtrees that contain no solution
- the number of such traversals can be exponential (e.g. TSP)

Parallelisation and speed up

Branch and bound is amenable to parallelisation:

- Maintain the global variables in **shared memory**
- Parallelise the traversal of the branches in any way you want

Shared memory needs to be **exclusively locked** only when updating the best current solution and bound (not when reading)

If t_{seq} and t_{par} refer to the time spent by the sequential and parallel versions of branch and bound, the **speed up** is

$$speedup = \frac{t_{seq}}{t_{par}}$$

Theoretically, $speedup \leq N$ when parallelisation is over N processors i.e. speed up is at most **linear in number of cores**

Observed super-linear speed ups

Sometimes parallelised DFBnB shows super-linear speed up

How come? Is the theory wrong?

Parallelised DFBnB explores tree more uniformly and finds good solutions quicker, which translates into **more pruning**

Theory isn't wrong: it says that speed up must be measured against **best sequential algorithm**

Conclusion: sequential DFBnB may be improved by randomly shuffling list of children at each node

Efficient depth-first branch and bound

As with IDA*, we can improve performance when the heuristic function is goal aware and the actions are invertible

In such case, we can exploit the recursion stack to get rid of the node data structure by maintaining a **single global state variable**

As seen in the previous slide, performance may also increase if the successor states of a node are explored in random order

General branch and bound

Branch and bound is a general technique. It can be used to find optimal configurations in combinatorial spaces

The model consists of:

- a finite configuration space S
- a function $g : S \to \mathbb{R}^{\geq 0}$ to be **minimized**
- a **bounding function** $h : 2^S \to \mathbb{R}^{\geq 0}$
- a **branching scheme** $b : 2^S \to 2^{2^S}$ where $b(S')$ is a partition of S' in more than 1 block

The idea is that for set $S' \subseteq S$ of configurations, the value $h(S')$ of the bounding function **lower bounds** the value $g(x)$ for each $x \in S'$

Let $S' \subseteq S$ and $b(S') = \{S_1, \ldots, S_n\}$. The branching **decomposes** the problem of finding a best configuration in S' into the subproblems of finding a best configuration in each of the S_i, $i = 1, \ldots, n$

General branch and bound: pruning

As before, the algorithm maintains the best current solution and prunes subproblems that cannot generate a better solution

General BnB explores a search tree where each node represents a subproblem $S' \subseteq S$:

- the root is the complete problem S
- the leaves are singletons $\{x\}$
- the children of $S' \subseteq S$ correspond to the subproblems in $b(S')$

During the traversal of the tree, a node S' is **pruned** when $f(S')$ is bigger than or equal to the cost of the best current solution

General branch and bound: pseudocode

```
1  % Branch and bound
2  Conf branch-and-bound(Subproblem R):
3      Conf best-solution = "some configuration in R"
4      unsigned best-bound = "cost of best-solution"
5
6      Queue q
7      q.insert(R)
8      while !q.empty()
9          Conf S = q.pop()
10
11         % test if subproblem can be pruned
12         if h(S) >= best-bound continue
13
14         % test whether we found a better solution
15         if S is singleton {x}
16             if g(x) < best-bound
17                 best-bound = g(x)
18                 best-solution = x
19             continue
20
21         % branch on subproblems
22         foreach S' in b(S)
23             q.insert(S')
24     return best-solution
```

General branch and bound: queue

Depending of the type of queue, one obtains different algorithms:

– if queue is LIFO queue, we obtain **depth-first branch and bound**

– if queue is min priority queue over the values of $h(\cdot)$, we obtain **best-first branch and bound (BFBnB)**

Summary

- DFBnB maintains an upper bound on optimal solution cost
- The search tree is explored in depth-first fashion, pruning branches that can't yield improvement on current solution
- Parellelisation and multi-threading may result in improved performance
- General branch and bound

Branch and bound for TSP

Goals for the lecture

- Naive depth-first branch and bound for TSP with MST heuristic
- Best-first branch and bound for TSP
- Different branching schemes and bounding functions

Naive DFBnB for TSP: search tree

We consider a naive of DFBnB for TSP whose search tree satisfies:

- branches correspond to partial tours; all tours begin at fixed city c_1
- the children of a node correspond to all cities not yet visited on the path for the node
- leaves correspond to all the $(n-1)!$ permutations on n cities whose first element is the city c_1

The g-value of a node n is the cost of the path corresponding to n

Symmetric TSP

Recall that a TSP instance consists of n cities together with a $n \times n$ matrix $D = (d_{ij})_{ij}$ of distance costs

The entry d_{ij} in D is the cost to go directly from city i to city j

We say that a TSP instance is **symmetric** if the D is symmetric; i.e. if $d_{ij} = d_{ji}$ for all cities i and j

The set of all symmetric TSP instances make up the **Symmetric TSP**

From now on, we **focus on Symmetric TSP**, yet the results can be adapted to the asymmetric case as well

Naive DFBnB for TSP: bounding

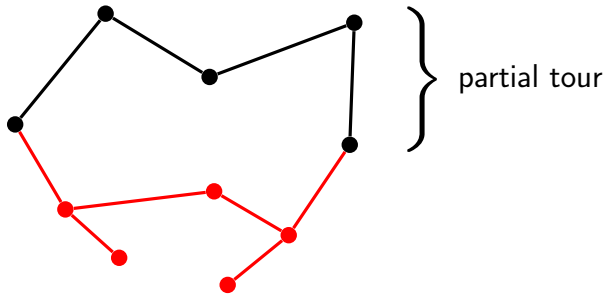
We use the MST heuristic for estimating the cost of completing a partial tour

Given a partial tour $\langle a_1, \ldots, a_k \rangle$, the MST heuristic is computed as:

1. Compute a MST T over $C \setminus \{a_1, \ldots, a_k\}$ with cost $c(T)$
2. Let e_1 and e_k be two edges of minimum cost that connect vertices a_1 and a_k to T respectively
3. The MST heuristic is $c(T) + c(e_1) + c(e_k)$

TSP: example of MST heuristic

TSP instance with 10 cities:



partial tour + MST

Naive DFBnB for TSP: analysis

The naive DFBnB for the TSP is a complete and optimal algorithm

However, it is only effective for small instances

For larger instances we need to use different **branching schemes**, **bounding functions**, and also consider other BnB algorithms

Branch and bound for TSP

TSP is one of the most studied problems in computer science

There are many algorithms, complete and approximate, for the TSP

Regarding BnB, there are many branching schemes and bounding functions. In the following, we study:

Branching schemes:

- simple edge splitting
- rule of Volgenant and Jonker [European J. of OR 9 83–89, 1982]

Bounding functions:

- computing minimum-cost edges at each vertex
- Held-Karp lower bound

Search tree

A node n in the search tree consists of two **disjoint** sets $I(n)$ and $E(n)$ of edges such that:

– all edges in $I(n)$ **belong** to all the solutions represented by the node n; these are the **included edges**

– all edges in $E(n)$ **do not belong** to any of the solutions represented by the node n; these are the **excluded edges**

Enforce invariants:

1. if $I(n)$ includes two edges incident at some vertex p, then $E(n)$ includes all other edges incident at p
2. if $I(n)$ contains a path, then $E(n)$ contains all edges that close the path into a sub-cycle

Branching by edge splitting

Consider a node $n = (I, E)$ in the search tree

Branching by edge splitting consists in selecting one edge $e \notin I \cup E$ and generating **two children** for n:

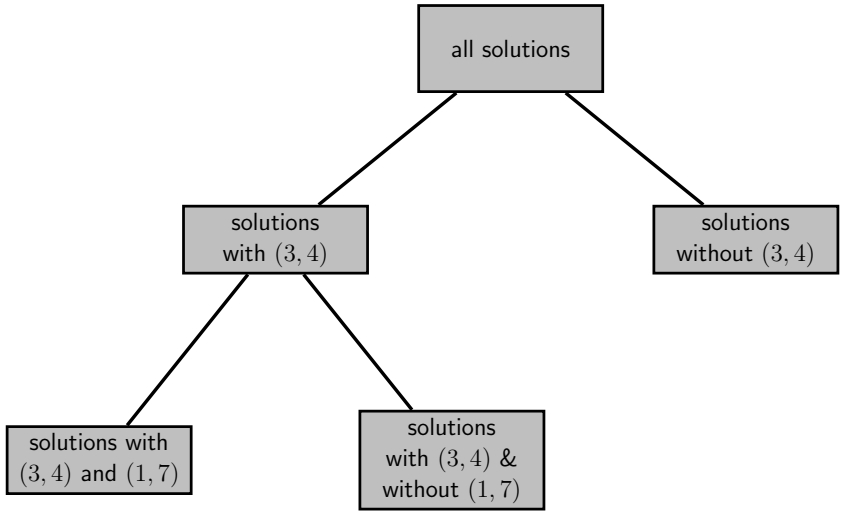
– left child $n_L = (I_L, E_L)$ that **includes edge e in the tours** (with $I_L = I \cup \{e\}$ and $E_L = E$)

– right child $n_R = (I_R, E_R)$ that **excludes edge e from the tours** (with $I_R = I$ and $E_R = E \cup \{e\}$)

– (in each case, previous invariants are enforced by possibly including/excluding more edges)

With edge splitting each internal node has exactly two children!

Edge splitting: example



Bounding: 1-trees

Let P be a TSP with n cities and $G = (V, E)$ be (undirected) K_n associated with P (i.e. costs in G given by costs in P)

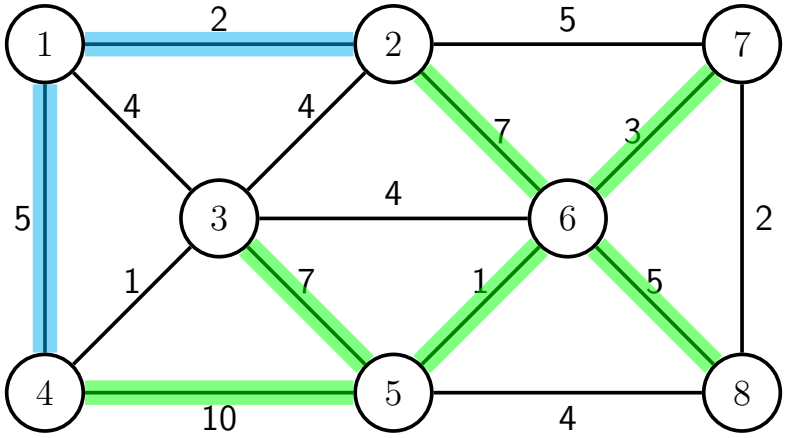
A **1-tree** is a connected subgraph H of G with n edges in which vertex 1 has degree 2 and belongs to the **unique cycle** in H (i.e. it is a tree plus an additional edge incident at vertex 1)

Alternatively, a 1-tree for G is the output of the following procedure:

- construct a spanning tree H for $G - \{1\}$
- choose two edges e_1 and e_2 incident at 1
- output $H + \{e_1, e_2\}$ (this is a subgraph on n vertices)

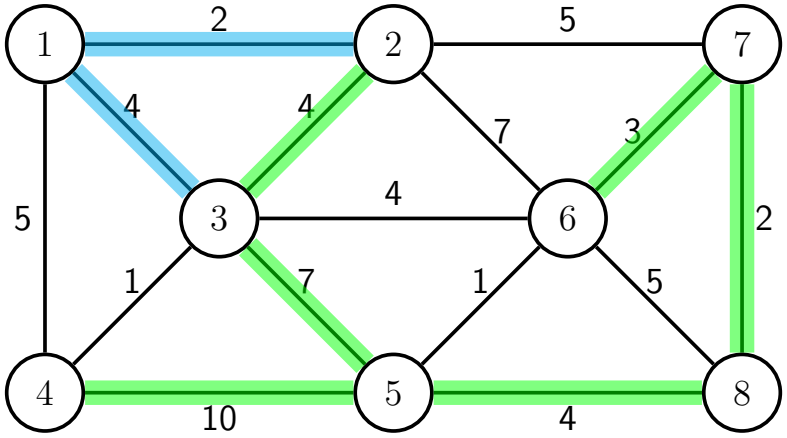
The cost of a 1-tree of minimum cost is a lower bound on the cost of any tour!

1-trees: example



cost = 40

1-trees: example



cost = 36

Computing 1-trees of minimum cost

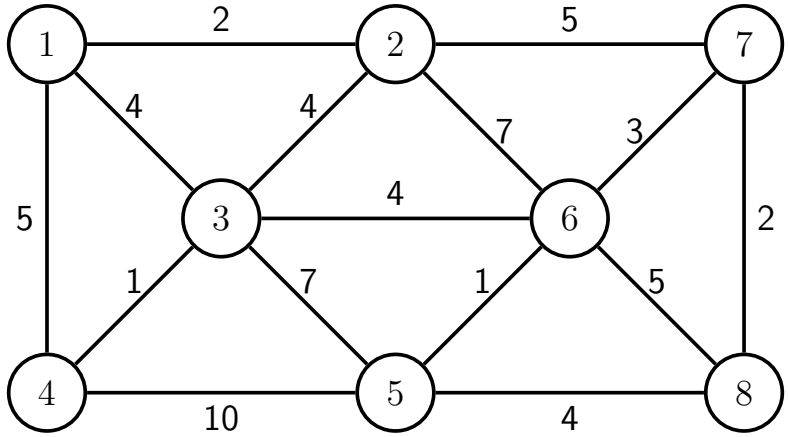
A **1-tree of minimum cost** can be obtained as follows:

- compute a MST H for $G - \{1\}$
- choose two edges e_1 and e_2 incident at 1 of **minimum cost**
- output $H + \{e_1, e_2\}$

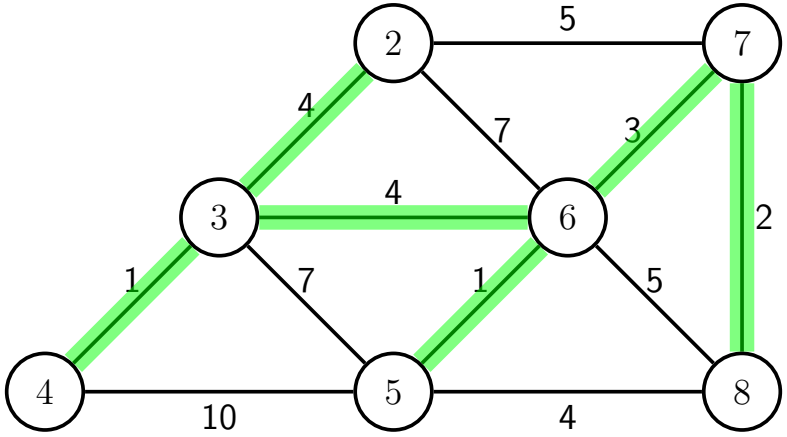
Finding the MST can be done in time:

- $O(E \log V)$ or $O(E \log^* V)$ using Kruskal's algorithm depending whether the edges need to be sorted or not
- $O(E + V \log V)$ using Prim's algorithm with a Fibonacci heap

Computing 1-trees of minimum cost: example

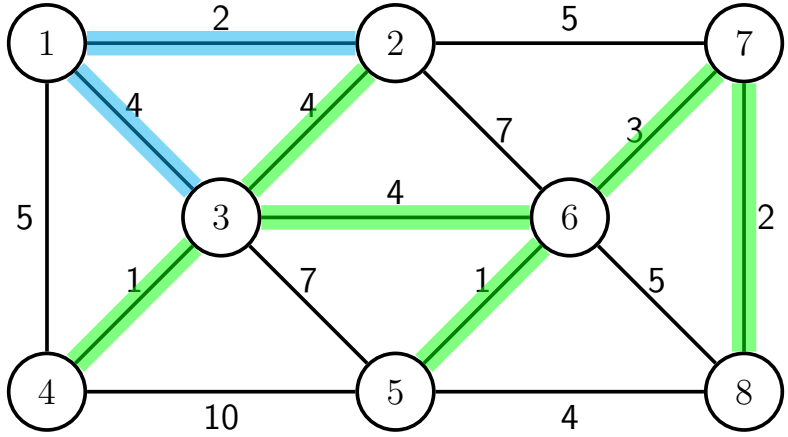


Computing 1-trees of minimum cost: example



cost = 15

Computing 1-trees of minimum cost: example



cost = 21

Computing 1-trees for node in search tree

For node (I, E) in search tree, 1-trees H relative to (I, E) satisfy:

- H contains all edges in I
- H contains no edge in E

A **1-tree of minimum cost** for node (I, E) can be computed by:

- assign ∞ cost to all edges in E
- compute a MST H for $G - \{1\}$ but assuming large negative costs to edges in I, forcing them to be included in H
- choose two edges e_1 and e_2 incident at 1 of **minimum cost**
- output $H + \{e_1, e_2\}$

Branching rule of Volgenant and Jonker

Let (I, E) be the current node and H be an optimal 1-tree for it

If every vertex has degree 2 in H, H is an **optimal TSP tour**

Else, there is vertex p with $\delta_H(p) > 2$ (because $\sum_{v \in V} \delta(v) = 2|E|$)

We claim that there are at least two edges e_1, e_2 in H that are both incident at p and not in $I \cup E$

Proof:

- (since $\delta_H(p) \geq 3$) if claim is false, there are 2 edges in H incident at p and that belong to $I \cup E$
- edges have cost $< \infty$ because they belong to H, thus none in E
- **contradiction** since if two edges belong to I, third edge incident at p must belong to E (by invariant) and cost of H would be ∞ □

Branching rule of Volgenant and Jonker

Let (I, E) be the current node and H be an optimal 1-tree for it

If every vertex has degree 2 in H, H is an **optimal TSP tour**

Else, there is vertex p with $\delta_H(p) > 2$ (because $\sum_{v \in V} \delta(v) = 2|E|$)

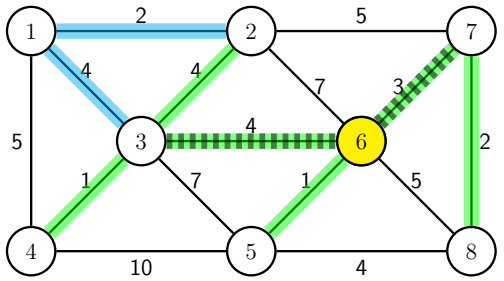
We claim that there are at least two edges e_1, e_2 in H that are both incident at p and not in $I \cup E$

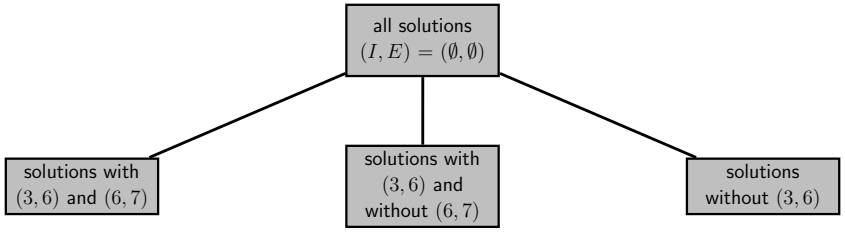
Volgenant and Jonker generate the following subproblems for (I, E):

- (I_1, E_1) with $I_1 = I \cup \{e_1, e_2\}$ and $E_1 = E$
- (I_2, E_2) with $I_2 = I \cup \{e_1\}$ and $E_2 = E \cup \{e_2\}$
- (I_3, E_3) with $I_3 = I$ and $E_3 = E \cup \{e_1\}$

Further, if p is incident at some edge in I, **don't generate** (I_1, E_1)

Branching by Volgenant and Jonker: example





Bounding by computing minimum-cost edges

A TSP tour contains two edges incident at each vertex

If we select two **minimum-cost** edges incident at each vertex, the half of the sum of the costs of selected edges is a lower bound on the cost of any tour

A lower bound for node (I, E) is obtained by assuring:

– each edge e in I is selected

– no edge e in E is selected

Improving the 1-tree lower bound

Let $D = (d_{ij})_{ij}$ be the cost matrix and let $\lambda = (\lambda_1, \ldots, \lambda_n)$ be a vector of numbers for each vertex i

Define **transformed distances** $D^\lambda = (d^\lambda_{ij})_{ij}$ as $d^\lambda_{ij} = d_{ij} + \lambda_i + \lambda_j$

The cost of a tour H wrt costs D^λ is $c_{D^\lambda}(H) = c_D(H) + 2\sum_i \lambda_i$

Let $T_1, T_2, \ldots$ be an **enumeration of all 1-trees** and let δ^k_i be the degree of vertex i in tree T_k

The cost of T_k wrt costs D^λ is $c_{D^\lambda}(T_k) = c_D(T_k) + \sum_i \delta^k_i \lambda_i$

Improving the 1-tree lower bound

For an optimal tour H^*,

$$c_{D^\lambda}(H^*) \;\geq\; \min_k c_{D^\lambda}(T_k) \;=\; \min_k\{c_D(T_k) + \textstyle\sum_k \delta^k_i \lambda_i\}$$

On the other hand, $c_{D^\lambda}(H^*) = c_D(H^*) + 2\sum_i \lambda_i$

Therefore, $c_D(H^*) \;\geq\; \min_k\{c_D(T_k) + \sum_k(\delta^k_i - 2)\lambda_i\} \;\doteq\; L(\lambda)$

For any λ, $L(\lambda)$ is **lower bound on the cost of an optimal tour** and can be computed with MST algorithm in polynomial time

The best lower bound is obtained by **maxing** $L(\lambda)$ over all λs

Held-Karp lower bound

The Held-Karp lower bound [Math. Prog. 1 6–25, 1971] is $L(\lambda^*)$ for the best vector λ^*

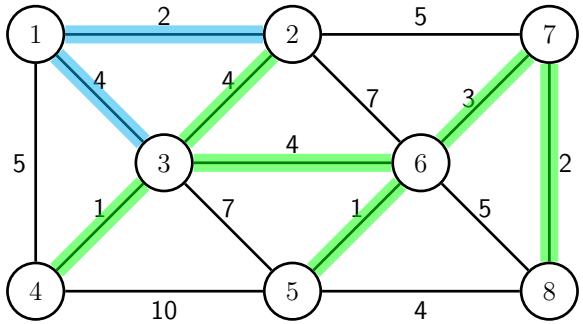
Finding λ^* is costly. Held and Karp propose a **subgradient method** to approximate λ^*:

1. Start with iterate $\lambda^0 = (0, \ldots, 0)$ and upper bound U
2. Let $k = 0$
3. Find optimal 1-tree T_k for costs D^{λ^k} and let $L = c_{D^{\lambda^k}}(T_k)$
4. If T_k is a tour or $L \geq U$, stop because an **optimal tour is found**
5. Let δ^k be the degrees of the vertices in T_k
6. Let $\lambda^{k+1}_i = \lambda^k_i + t^k(\delta^k_i - 2)$ where the step length t^k is

$$t^k = 2(U - L) \big/ \textstyle\sum_j(\delta^k_j - 2)^2$$

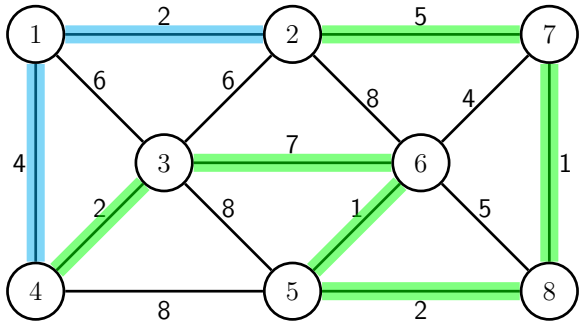
   (**Invariant:** $\sum_i \lambda^{k+1}_i = \sum_i \lambda^k_i$)
7. If $k <$ MaxIterations, increase $k = k + 1$ and go to 3

Held-Karp lower bound: example



Initially:	$k = 0$:	
$U = 25$	(reweighted graph)	
$\lambda^0 = (0, 0, 0, 0, 0, 0, 0, 0)$	(optimal 1-tree)	
	$L = 21$	
	$\delta^0 = (2, 2, 4, 1, 1, 3, 2, 1)$	
	$t^0 = 2(25 - 21)/8 = 1$	
	$\lambda^1 = (0, 0, 2, -1, -1, 1, 0, -1)$	

Held-Karp lower bound: example



Initially:	$k = 0$:	$k = 1$:
$U = 25$	(reweighted graph)	(reweighted graph)
$\lambda^0 = (0, 0, 0, 0, 0, 0, 0, 0)$	(optimal 1-tree)	(optimal 1-tree)
	$L = 21$	$L = 24$
	$\delta^0 = (2, 2, 4, 1, 1, 3, 2, 1)$	**(stop)**
	$t^0 = 2(25 - 21)/8 = 1$	
	$\lambda^1 = (0, 0, 2, -1, -1, 1, 0, -1)$	

Plugging Held-Karp lower bound in BnB

Given node (I, E) and best current solution with cost U:

- Compute Held-Karp lower bound and best 1-tree T using U
- If T is tour, T is an optimal TSP tour
- If cost of T is equal or bigger than U, prune branch because there is no better solution below current node
- Apply branching scheme

Summary

- Naive DFBnB for TSP
- Better versions achieved using general branch and bound algorithm with different branching schemes and bounding functions
- Lower bounds can be further improved using "cutting planes". Resulting algorithm is called **branch and cut**