Data Structures and Algorithms

11. Heuristic Search Algorithms

Basic Stuff You're Gonna Need to Search over a Graph
Where To Search Next?

Nir Lipovetzky

THE UNIVERSITY OF MELBOURNE

Path-finding in graphs

Search algorithms over **directed graphs**:

- The **search nodes** (vertex) of the graph represent some information
- The edges (v, v') capture transitions

One way to understand search algorithms is to think about **path-finding** over **graphs**.

Classification of Search Algorithms

Blind search vs. heuristic (or informed) search:

- Blind search algorithms: Only use the basic ingredients for general search algorithms.
  - *e.g., Depth First Search (DFS), Breadth-first search (BrFS), Uniform Cost (Dijkstra), Iterative Deepening (ID)*
- Heuristic search algorithms: Additionally use heuristic functions which estimate the distance (or remaining cost) to reach an target vertex.
  - *e.g., A*, IDA*, Hill Climbing, Best First, WA*, DFS B&B, LRTA*, . . .*

Systematic search vs. local search:

- Systematic search algorithms: Consider a large number of search nodes simultaneously.
- Local search algorithms: Work with one (or a few) candidate solutions (search nodes) at a time.

$\rightarrow$ This is not a black-and-white distinction; there are *crossbreeds* (e.g., enforced hill-climbing).

Before We Begin

Blind search vs. informed search:

- Blind search does not require any input beyond the graph.

  → Pros and Cons? Pro: No additional work for the programmer. Con: It's not called "blind" for nothing . . . same expansion order regardless what the problem actually is.

- Informed search requires as additional input a heuristic function h that maps nodes to estimates of their distance.

  → Pros and Cons? Pro: Typically more effective in practice. Con: Somebody's gotta come up with/implement h.

Before We Begin, ctd.

Blind search strategies we'll discuss:

- Breadth-first search. Advantage: time complexity.
  Variant: Uniform cost search.
- Depth-first search. Advantage: space complexity.
- Iterative deepening search. Combines advantages of breadth-first search and depth-first search. Uses depth-limited search as a sub-procedure.

Heuristic Search Algorithms: Systematic

→ Heuristic search algorithms are the most common and overall most successful algorithms for path-finding in graphs.

Systematic heuristic search algorithms:

- Greedy best-first search.
- Weighted A^*.
- A^*.
- IDA^*, depth-first branch-and-bound search, ...

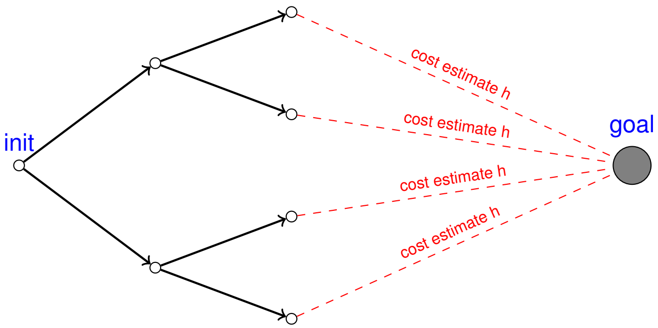
Heuristic Search Algorithms: Local

$\rightarrow$ Heuristic search algorithms are the most common and overall most successful algorithms for path-finding.

Local heuristic search algorithms:

- Hill-climbing.
- Beam search, tabu search, genetic algorithms, simulated annealing, ...

Heuristic Search: Basic Idea



→ Heuristic function h estimates the cost of an optimal path to the goal; search gives a preference to explore states with small h.

Heuristic Functions

Heuristic searches require a heuristic function to estimate remaining cost:

Definition (Heuristic Function). A heuristic function, short heuristic, is a function $h : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$. Its value $h(s)$ for a state s is referred to as the state's heuristic value, or h-value.

Definition (Remaining Cost, h^*). For a state $s \in S$, the state's remaining cost is the cost of a shortest path from s to a goal state g , or ∞ if there exists no path for s. The perfect heuristic , written h^*, assigns every $s \in S$ its remaining cost as the heuristic value.

$\rightarrow$ Recall that a state s is equivalent to a vertex v in the graph!

Heuristic Functions: Discussion

What does it mean to "estimate remaining cost"?

- For many heuristic search algorithms, h does not need to have any properties for the algorithm to "work" (= be correct and complete).

  $\rightarrow$ h is *any* function from states to numbers . . .
- Search performance depends crucially on "how well h reflects h^*"!!

  $\rightarrow$ This is informally called the informedness or quality of h.
- For some search algorithms, like A^*, we can *prove* relationships between formal quality properties of h and search efficiency (mainly the number of expanded nodes).
- For other search algorithms, "it works well in practice" is often as good an analysis as one gets.

Heuristic Functions: Discussion, ctd.

"Search performance depends crucially on the informedness of h ..."

Any other property of h that search performance crucially depends on?

"... and on the computational overhead of computing h!!"

Extreme cases:

- $h = h^*$: Perfectly informed; computing it = solving the original problem in the first place.
- $h = 0$: No information at all; can be "computed" in constant time.

$\rightarrow$ Successful heuristic search requires a good trade-off between h's informedness and the computational overhead of computing it.

$\rightarrow$ This really is what research is all about! Devise methods that yield good estimates at reasonable computational costs.

Properties of Heuristic Functions

Definition. The heuristic is called:

- Admissible if $h(s) \leq h^*(s)$ for all $s \in S$;

Other properties not discussed in this course: Goal-Aware, Consistent and Safe.

Motivation: For You

Imagine . . .

You have completed your studies, and have been hired by some company in the software-making business.

Boss: *Hey you, here's that problem. Solve it.*

You (thinking): *Hm, I think heuristic search might work.*

You (thinking): *Hm, I need a heuristic function. How?!?*

→ "Relax"ing is a methodology to construct heuristic functions.

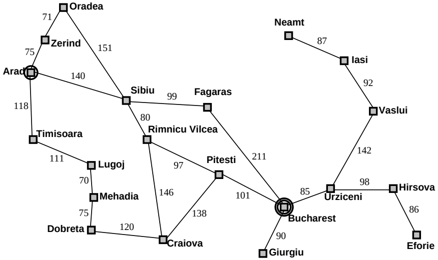
How to Relax Informally

How To Relax:

- You have a problem, $\mathcal{P}$, whose perfect heuristic h^* you wish to estimate.
- You define a simpler problem, $\mathcal{P}'$, whose perfect heuristic h'^* can be used to estimate h^*.
- You define a transformation, r, that simplifies instances from $\mathcal{P}$ into instances $\mathcal{P}'$.
- Given $\Pi \in \mathcal{P}$, you estimate $h^*(\Pi)$ by $h'^*(r(\Pi))$.

$\rightarrow$ Relaxation means to simplify the problem, and take the solution to the simpler problem as the heuristic estimate for the solution to the actual problem.

Relaxation in Route-Finding



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

How to derive straight-line distance by relaxation?

- Problem $\mathcal{P}$: Route finding.
- Simpler problem $\mathcal{P}'$: Route finding for birds.
- Perfect heuristic h'^* for $\mathcal{P}'$: Straight-line distance.
- Transformation r: Pretend you're a bird.

Greedy Best-First Search

Greedy Best-First Search (with duplicate detection)

open $:=$ **new** priority queue ordered by ascending $h(\mathit{state}(\sigma))$
open.insert(make-root-node(init()))
closed $:= \emptyset$
while not *open*.empty():
  $\sigma :=$ *open*.pop-min() /* get best state */
  if $\mathit{state}(\sigma) \notin \mathit{closed}$: /* check duplicates */
    $\mathit{closed} := \mathit{closed} \cup \{\mathit{state}(\sigma)\}$ /* close state */
    if is-goal(state(σ)): **return** extract-solution(σ)
    for each $(s') \in$ succ($\mathit{state}(\sigma)$): /* expand state */
      $\sigma' :=$ make-node(σ, s')
      if $h(\mathit{state}(\sigma')) < \infty$: *open*.insert($\sigma'$): **return** unsolvable

Greedy Best-First Search: Remarks

Properties:

- Complete? Yes (Due to duplicate detection.)
- Optimal? No.[1]

Implementation:

- Priority queue: e.g., a min heap.
- "Check Duplicates": Could already do in "expand state"; done here after "get best state" *only* to more clearly point out relation to A^*.

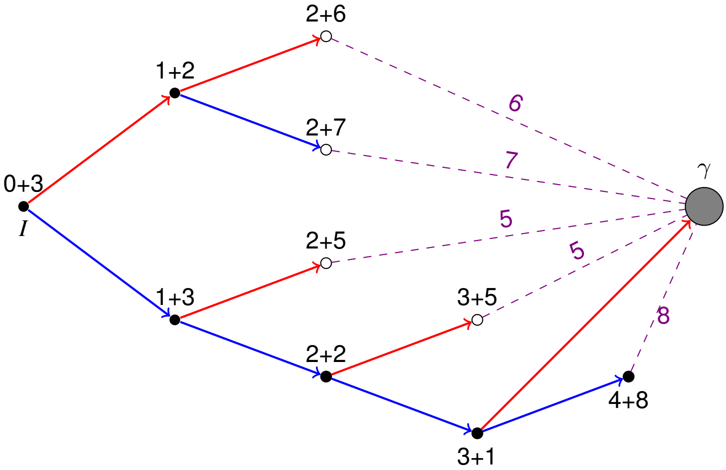
[1] Even for perfect heuristics! E.g., say the start state has two transitions to goal states, one of which costs a million bucks while the other one is free. Nothing keeps Greedy Best-First Search from choosing the bad one.

A^*

A^* (with duplicate detection and re-opening)

open $:=$ **new** priority queue ordered by ascending $g(\textit{state}(\sigma)) + h(\textit{state}(\sigma))$
open.insert(make-root-node(init()))
closed $:= \emptyset$
best-g $:= \emptyset$/* maps states to numbers */
while not *open*.empty():
    $\sigma :=$ *open*.pop-min()
    if $\textit{state}(\sigma) \notin \textit{closed}$ **or** $g(\sigma) < \textit{best-g}(\textit{state}(\sigma))$:
      /* re-open if better g; note that all σ' with same state but worse g
        are *behind* σ in *open*, and will be skipped when their turn comes */
      closed $:= \textit{closed} \cup \{\textit{state}(\sigma)\}$
      $\textit{best-g}(\textit{state}(\sigma)) := g(\sigma)$
      if is-goal(state(σ)): **return** extract-solution(σ)
      for each $(s') \in$ succ($\textit{state}(\sigma)$):
        $\sigma' :=$ make-node(σ, s')
        if $h(\textit{state}(\sigma')) < \infty$: *open*.insert$(\sigma')$
return unsolvable

A^*: Example



A^*: Terminology

- f-value of a state: defined by $f(s) := g(s) + h(s)$.
- Generated nodes: Nodes inserted into *open* at some point.
- Expanded nodes: Nodes σ popped from *open* for which the test against *closed* and *distance* succeeds.
- Re-expanded nodes: Expanded nodes for which *state*$(\sigma) \in$ *closed* upon expansion (also called re-opened nodes).

A^*: Remarks

Properties:

- Complete? Yes (Even without duplicate detection.)
- Optimal? Yes, for admissible heuristics. (Even without duplicate detection.)

Implementation:

- Popular method: break ties ($f(s) = f(s')$) by smaller h-value.
- Common, hard to spot bug: check duplicates at the wrong point.
- Our implementation is optimized for readability not for efficiency!

Hill-Climbing

Hill-Climbing

$\sigma :=$ make-root-node(init())
forever:
  if is-goal(state(σ)):
    return extract-solution(σ)
  $\Sigma' :=$ { make-node(σ, s') | $(s') \in$ succ(state(σ)) }
  $\sigma :=$ an element of Σ' minimizing h /* (random tie breaking) */

Remarks:

- Is this complete or optimal? No.
- Can easily get stuck in local minima where immediate improvements of $h(\sigma)$ are not possible.
- Many variations: tie-breaking strategies, restarts, . . .

Properties of Search Algorithms

ID: Iterative Deepening (not included in this lecture)

	DFS	BrFS	ID	A*	HC
Complete	No	Yes	Yes	Yes	No
Optimal	No	Yes*	Yes	Yes	No
Time	∞	b^d	b^d	b^d	∞
Space	$b \cdot d$	b^d	$b \cdot d$	b^d	b

- Parameters: d is solution depth; b is branching factor
- Breadth First Search (BrFS) optimal when costs are uniform
- A* optimal when h is **admissible**; $h \leq h^*$
- Searh Algorithms visualization: https://www.youtube.com/watch?v=rZHtHJlJa2w
- More Search Algorithms visualization: http://www.redblobgames.com/pathfinding/a-star/introduction.html
- Interactive Grid Path Finding visualization: https://qiao.github.io/PathFinding.js/visual/

Applications of Path finding

Many problems can be understood and solved as path-finding search problems over graphs!

- Google self driving car
  `https://youtu.be/qXZt-B7iUyw`
- Computer Games
  `https://www.youtube.com/watch?v=DlkMs4ZHHr8`
- Base algorithm (A^*) running “Best AI in Computer Games”
  `https://youtu.be/1OXb8mg9IVw?t=7m45s`
- Local search, Evolution
  `https://www.youtube.com/watch?v=bBt0imn77Zg`

Questionnaire, ctd.

Question!

Is informed search always better than blind search?

(A): Yes. (B): No.

→ In greedy best-first search, the heuristic may yield larger search spaces than uniform-cost search. E.g., in path finding, say you want to go from Melbourne to Sydney, but $h(\text{Perth}) < h(\text{Canberra})$.

→ In A^* with an admissible heuristic and duplicate checking, we cannot do worse than uniform-cost search: $h(s) > 0$ can only reduce the number of states we must consider to prove optimality.

→ Also, in the above example, A^* doesn't expand Perth with *any* admissible heuristic, because $g(\text{Perth}) > g(\text{Sydney})$!

→ "Trusting the heuristic" has its dangers! Sometimes g helps to reduce search.

Summary

Search algorithms mainly differ in order of node expansion:

- Blind vs. heuristic (or informed) search.
- Systematic vs. local search.

Summary (ctd.)

- Search strategies differ (amongst others) in the order in which they expand search nodes, and in the way they use duplicate elimination. Criteria for evaluating them are completeness, optimality, time complexity, and space complexity.
- Breadth-first search is optimal but uses exponential space; depth-first search uses linear space but is not optimal. Iterative deepening search combines the virtues of both.

Summary (ctd.)

Heuristic Functions: Estimators for remaining cost.

- Usually: The more informed, the better performance.
- Desiderata: admissible, informed
- The ideal: Perfect heuristic h^*.

Heuristic Search Algorithms:

- Most common algorithms if you don't care about optimality:
  - Greedy best-first search.
  - Weighted A^*.
  - hill-climbing.
- Most common algorithm if you care about optimality:
  - A^*.
  - IDA^*.

Search Algorithms for Path Finding in Directed Graphs

- **Blind search/Brute force algorithms**
  - Goal plays **passive** role in the search
  - *e.g., Depth First Search (DFS), Breadth-first search (BrFS), Uniform Cost (Dijkstra), Iterative Deepening (ID)*
- **Informed/Heuristic Search Algorithms**
  - Goals plays **active** role in the search through **heuristic function** $h(s)$ that estimates cost from s to the goal
  - *e.g., A*, IDA*, Hill Climbing, Best First, WA*, DFS B&B, LRTA*, . . .*