Informed Search

Chapter 3.5 – 3.6, 4.1

Informed Search

- Informed searches use domain knowledge to guide selection of the best path to continue searching
- Heuristics are used, which are informed guesses
- Heuristic means "serving to aid discovery"

Informed Search

- Define a heuristic function, ***h*(*n*)**
  - uses domain-specific info. in some way
  - is computable from the current state description
  - it estimates
    - the "goodness" of node ***n***
    - how close node ***n*** is to a goal
    - the cost of *minimal* cost path from node ***n*** to a goal state

Informed Search

- $h(n) \geq 0$ for all nodes n
- $h(n) = 0$ implies that n is a goal node
- $h(n) = \infty$ implies that n is a dead end from which a goal cannot be reached
- All domain knowledge used in the search is encoded in the heuristic function, h
- An example of a *"weak method"* for AI because of the limited way that domain-specific information is used to solve a problem

Best-First Search

- Sort nodes in the Frontier list by increasing values of an evaluation function, ***f*(*n*)**, that incorporates domain-specific information
- This is a generic way of referring to the class of informed search methods

Greedy Best-First Search

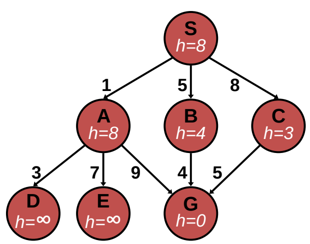
- Use as an evaluation function, ***f*(*n*) = *h*(*n*)**, sorting nodes in the Frontier list by increasing values of ***f***
- Selects the node to expand that is believed to be closest (i.e., smallest ***f*** value) to a goal node

Greedy Best-First Search

***f*(*n*) = *h*(*n*)**

# of nodes tested: 0, expanded: 0

expnd. node	Frontier list
	{S:8}

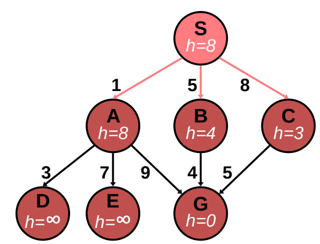


Greedy Best-First Search

***f*(*n*) = *h*(*n*)**

# of nodes tested: 1, expanded: 1

expnd. node	Frontier list
	{S:8}
S not goal	{C:3,B:4,A:8}

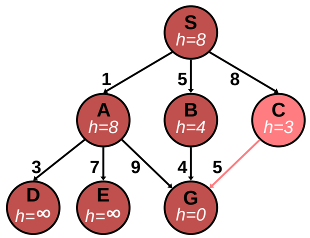


Greedy Best-First Search

$f(n) = h(n)$

# of nodes tested: 2, expanded: 2

expnd. node	Frontier list
	{S:8}
S	{C:3,B:4,A:8}
C not goal	{G:0,B:4,A:8}

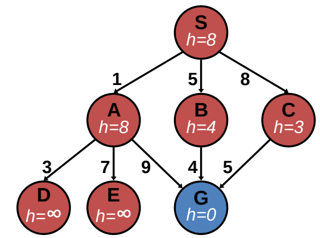


Greedy Best-First Search

$f(n) = h(n)$

# of nodes tested: 3, expanded: 2

expnd. node	Frontier list
	{S:8}
S	{C:3,B:4,A:8}
C	{G:0,B:4, A:8}
G goal	{B:4, A:8} no expand

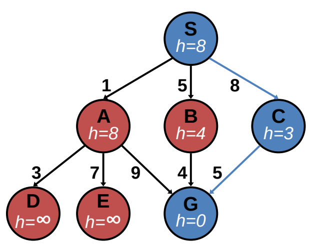


Greedy Best-First Search

$f(n) = h(n)$

# of nodes tested: 3, expanded: 2

expnd. node	Frontier list
	{S:8}
S	{C:3,B:4,A:8}
C	{G:0,B:4, A:8}
G	{B:4, A:8}



- *Fast but not optimal*

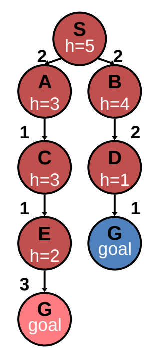
path: S,C,G
cost: 13

Greedy Best-First Search

- Not complete
- Not optimal/admissible

Greedy search finds the left goal (solution cost of 7)

Optimal solution is the path to the right goal (solution cost of 5)



Beam Search

- Use an evaluation function $f(n) = h(n)$ as in greedy best-first search, but restrict the maximum size of the Frontier list to a constant, k
- Only keep k best nodes as candidates for expansion, and throw away the rest
- More space efficient than Greedy Search, but may throw away a node on a solution path
- Not complete
- Not optimal/admissible

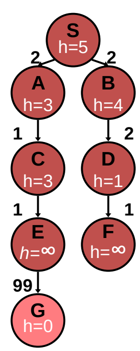
Algorithm A Search

- Use as an evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is minimal cost path from start to current node n (as defined in UCS)
- The g term adds a "breadth-first component" to the evaluation function
- Nodes on the *Frontier* are ranked by the *estimated cost of a solution*, where $g(n)$ is the cost from the start node to node n, and $h(n)$ is the estimated cost from node n to a goal

Algorithm A Search

- Not complete
- **Not optimal/admissible**

Algorithm A never expands E because h(E) = ∞



Algorithm A* Search

- Use the same evaluation function used by Algorithm A, except add the constraint that for *all* nodes n in the search space, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost of the minimal cost path from n to a goal
- The cost to the nearest goal is **never *over-estimated***
- When $h(n) \leq h^*(n)$ holds true for *all* n, h is called an **admissible heuristic function**
- An admissible heuristic guarantees that a node on the optimal path cannot look so bad so that it is never considered

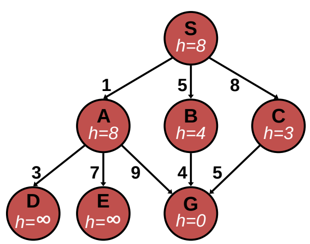
Admissible Heuristics are Good for Playing *The Price is Right*

Algorithm A* Search

- Complete
- Optimal / Admissible

Example

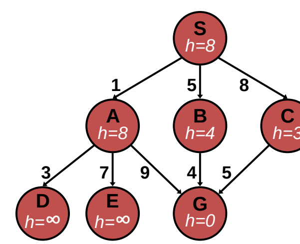
n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S				
A				
B				
C				
D				
E				
G				



g(n) = actual cost to get to node *n* from start

Example

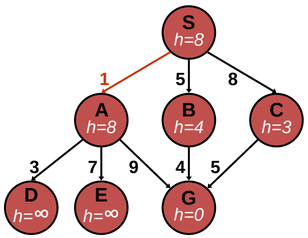
n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0			
A				
B				
C				
D				
E				
G				



g(n) = actual cost to get to node *n* from start

Example

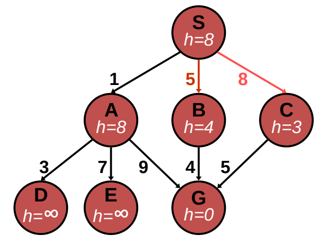
n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0			
A	1			
B				
C				
D				
E				
G				



g(n) **= actual cost to get to node *n* from start**

Example

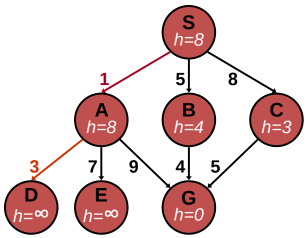
n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0			
A	1			
B	5			
C	8			
D				
E				
G				



g(n) **= actual cost to get to node *n* from start**

Example

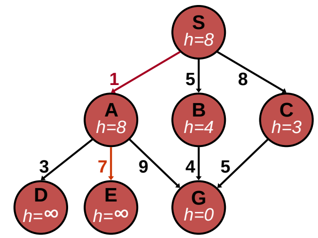
n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0			
A	1			
B	5			
C	8			
D	1+3 = 4			
E				
G				



g(n) **= actual cost to get to node *n* from start**

Example

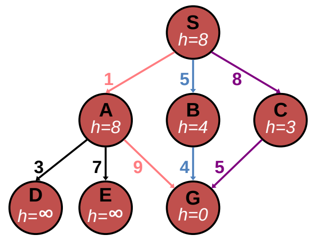
n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0			
A	1			
B	5			
C	8			
D	4			
E	1+7 = 8			
G				



g(n) **= actual cost to get to node *n* from start**

Example

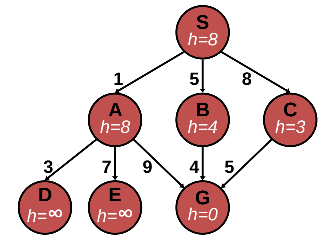
n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0			
A	1			
B	5			
C	8			
D	4			
E	8			
G	10/9/13			



g*(*n*)** = actual cost to get to node ***n from start

Example

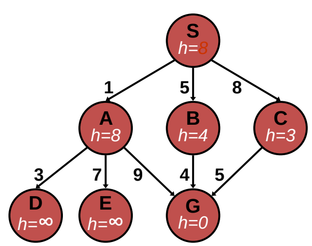
n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0			
A	1			
B	5			
C	8			
D	4			
E	8			
G	10/9/13			



h*(*n*)** = estimated cost to get to a goal from node ***n

Example

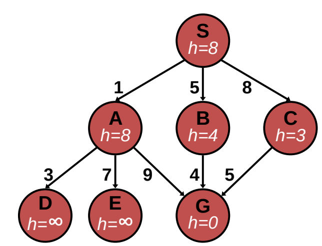
n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0	8		
A	1	8		
B	5	4		
C	8	3		
D	4	∞		
E	8	∞		
G	10/9/13	0		



h*(*n*)** = estimated cost to get to a goal from node ***n

Example

n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0	8	8	
A	1	8	9	
B	5	4	9	
C	8	3	11	
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	

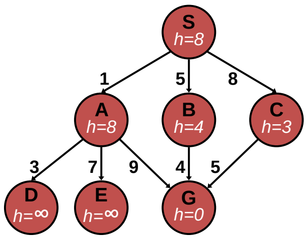


$f(n) = g(n) + h(n)$

actual cost to get from start to *n* plus estimated cost from *n* to goal

Example

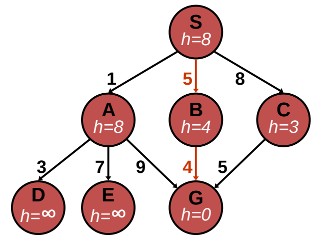
n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0	8	8	
A	1	8	9	
B	5	4	9	
C	8	3	11	
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	



h*(n) **= true cost of minimal path from *n* to a goal**

Example

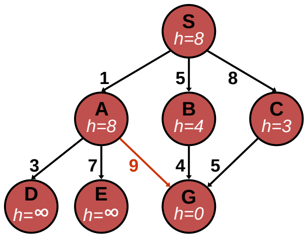
n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0	8	8	9
A	1	8	9	
B	5	4	9	
C	8	3	11	
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	



h*(n) **= true cost of minimal path from *n* to a goal**

Example

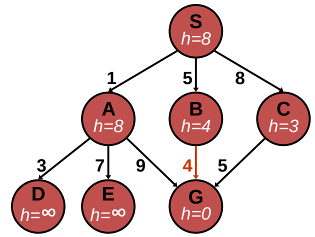
n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0	8	8	9
A	1	8	9	9
B	5	4	9	
C	8	3	11	
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	



h*(n) **= true cost of minimal path from *n* to a goal**

Example

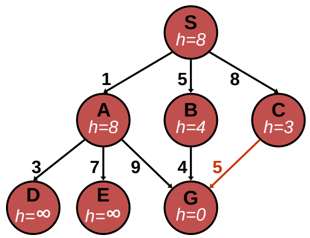
n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	



h*(n) **= true cost of minimal path from *n* to a goal**

Example

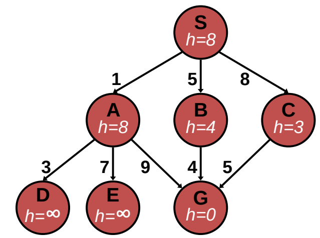
n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	



$h^*(n)$ = true cost of minimal path from n to a goal

Example

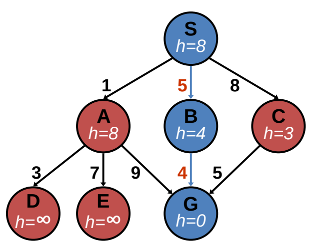
n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	∞	∞	∞
E	8	∞	∞	∞
G	10/9/13	0	10/9/13	0



$h^*(n)$ = true cost of minimal path from n to a goal

Example

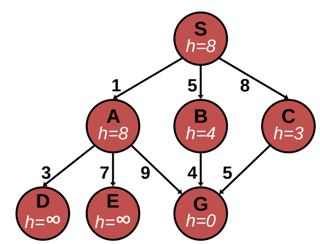
n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	∞	∞	∞
E	8	∞	∞	∞
G	10/9/13	0	10/9/13	0



optimal path = S,B,G
cost = 9

Example

n	*g(n)*	*h(n)*	*f(n)*	*h*(n)*
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	∞	∞	∞
E	8	∞	∞	∞
G	10/9/13	0	10/9/13	0



Since $h(n) \leq h^*(n)$ for all n, h is admissible

Admissible Heuristic Functions, *h*

- 8-Puzzle example

Example State

1		5
2	6	3
7	4	8

Goal State

1	2	3
4	5	6
7	8	

- Which of the following are admissible heuristics?

$h(n)$ = number of tiles in wrong position
$h(n) = 0$
$h(n) = 1$
$h(n)$ = sum of "City-block distance" between each tile and its goal location

Note: City-block distance = L_1 norm

Admissible Heuristic Functions, *h*

Which of the following are admissible heuristics?

$h(n) = h^*(n)$

$h(n) = \max(2, h^*(n))$

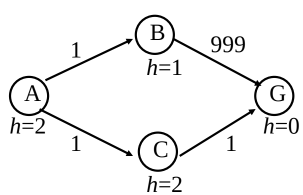
$h(n) = \min(2, h^*(n))$

$h(n) = h^*(n) - 2$

$h(n) = \sqrt{h^*(n)}$

When should A* Stop?

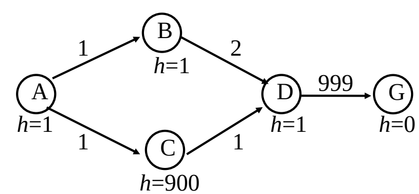
- A* should terminate only when a goal is popped from the priority queue



- Same rule as for uniform cost search
- A* with $h() = 0$ ***is*** uniform cost search

A* Revisiting Expanded States

- One more complication: A* can revisit an expanded state (on *Frontier* **or** *Expanded*), and discover a better path



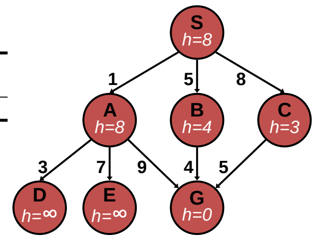
- Solution: Put *D* back into the priority queue, with the smaller *g* value

A* Search

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

# of nodes tested: 0, expanded: 0

expnd. node	Frontier list
	{S:0+8}

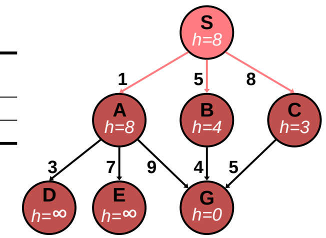


A* Search

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

# of nodes tested: 1, expanded: 1

expnd. node	Frontier list
	{S:8}
S not goal	{A:1+8,B:5+4,C:8+3}

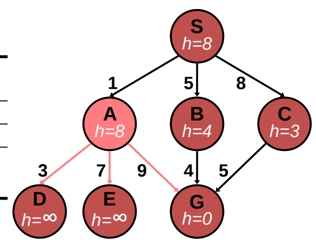


A* Search

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

# of nodes tested: 2, expanded: 2

expnd. node	Frontier list
	{S:8}
S	{A:9,B:9,C:11}
A not goal	{B:9,G:1+9+0,C:11, D:1+3+∞,E:1+7+∞}

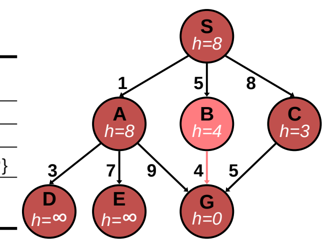


A* Search

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

# of nodes tested: 3, expanded: 3

expnd. node	Frontier list
	{S:8}
S	{A:9,B:9,C:11}
A	{B:9,G:10,C:11,D:∞,E:∞}
B not goal	{G:5+4+0,~~G:10~~,C:11, D:∞,E:∞} replace

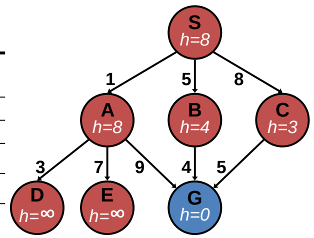


A* Search

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

\# of nodes tested: 4, expanded: 3

expnd. node	Frontier list
	{S:8}
S	{A:9,B:9,C:11}
A	{B:9,G:10,C:11,D:∞,E:∞}
B	{G:9,C:11,D:∞,E:∞}
G goal	{C:11,D:∞,E:∞} not expanded

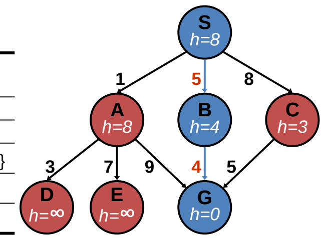


A* Search

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

\# of nodes tested: 4, expanded: 3

expnd. node	Frontier list
	{S:8}
S	{A:9,B:9,C:11}
A	{B:9,G:10,C:11,D:∞,E:∞}
B	{G:9,C:11,D:∞,E:∞}
G	{G:10,C:11,D:∞,E:∞}



- *Pretty fast and optimal*

path: S,B,G
cost: 9

Proof of A* Optimality (by Contradiction)

- Let

  G be the goal in the optimal solution

  $G2$ be a sub-optimal goal

  f^* be the cost of the optimal path from Start to G

  $g(G2) > f^*$ and assume $G2$ is found using A* where $f(n) = g(n) + h(n)$, and $h(n)$ is admissible
- That is, *A** found a sub-optimal path (which it shouldn't)

Proof of A* Optimality (by Contradiction)

- Let n be some node on the *optimal* path but *not* on the path to $G2$
- $f(n) \leq f^*$

  by admissibility, since $f(n)$ never overestimates the cost to the goal it must be ≤ the cost of the optimal path
- $f(G2) \leq f(n)$

  $G2$ was chosen over n for the sub-optimal goal to be found
- $f(G2) \leq f^*$

  combining equations

Proof of A* Optimality (by Contradiction)

- $f(G2) \leq f^*$
- $g(G2) + h(G2) \leq f^*$

  substituting the definition of f
- $g(G2) \leq f^*$

  $h(G2) = 0$ since $G2$ is a goal node
- This contradicts the assumption that G2 was sub-optimal, $g(G2) > f^*$
- Therefore, A* is optimal with respect to path cost; A* search never finds a sub-optimal goal

A* : The Dark Side

- A* can use lots of memory:

  O(number of states)
- For really big search spaces, A* will run out of memory

Devising Heuristics

Are often defined by relaxing the problem, i.e., computing exact cost of a solution to a *simplified* version of problem

- remove constraints: 8-puzzle movement
- simplify problem: straight line distance for 8-puzzle and mazes

Comparing Iterative Deepening with A*

[from Russell and Norvig, page 104, Fig 3.29]

	For 8-puzzle, average number of states expanded over 100 randomly chosen problems in which optimal path is length …		
	… 4 steps	… 8 steps	… 12 steps
Depth-First Iterative Deepening	**112**	**6,300**	**3.6×10^6**
A* search using "number of misplaced tiles" as the heuristic	**13**	**39**	**227**
A* using "Sum of Manhattan distances" as the heuristic	**12**	**25**	**73**

Devising Heuristics

- Goal of an admissible heuristic is to get as close to the actual cost without going over
- Must also be relatively fast to compute
- Trade off:
  use more time to compute a complex heuristic versus use more time to expand more nodes with a simpler heuristic

Devising Heuristics

- If $h(n) = h^*(n)$ for all n,
  - only nodes on optimal solution path are expanded
  - no unnecessary work is performed
- If $h(n) = 0$ for all n,
  - the heuristic is admissible
  - A* performs exactly as Uniform-Cost Search (UCS)
- The closer h is to h^*,
  the fewer extra nodes that will be expanded

Devising Heuristics

If $h1(n) \leq h2(n) \leq h^*(n)$ for all n that aren't goals, then $h2$ dominates $h1$

- $h2$ is a *better heuristic* than $h1$
- A* using $h1$ (i.e., A1*) expands *at least as many* if not more nodes than using A* with $h2$ (i.e., A2*)
- A2* is said to be better informed than A1*

Devising Heuristics

For an admissible heuristic

- h is frequently very simple
- therefore search resorts to (almost) UCS through parts of the search space

Devising Heuristics

- If optimality is *not* required, i.e., **satisficing solution** okay, then
- Goal of heuristic is then to get as close as possible, either under or over, to the actual cost
- It results in many fewer nodes being expanded than using a poor, but provably admissible, heuristic

Devising Heuristics

A* often suffers because it cannot venture down a single path unless it is almost continuously having success (i.e., h is decreasing); any failure to decrease h will almost immediately cause the search to switch to another path

Local Searching

- Systematic searching: search for a **path** from start state to a goal state, then "execute" solution path's sequence of operators
  - BFS, DFS, IDS, UCS, Greedy Best-First, A, A*, etc.
  - **ok** for small search spaces
  - **not okay** for NP-Hard problems requiring exponential time to find the (optimal) solution

Optimization Problems

- **Now a different setting**:
  - Each state s has a score or cost, $f(s)$, that we can compute
  - The goal is to find the state with the highest (or lowest) score, or a reasonably high (low) score
  - We do ***not*** care about the path
  - This is an optimization problem
  - **Enumerating the states is intractable**
  - Previous search algorithms are too expensive
  - No known algorithm for finding optimal solution efficiently

Traveling Salesperson Problem (TSP)

- Classic NP-Hard problem:
  A salesperson wants to visit a list of cities
  - stopping in each city only *once*
  - returning to the first city
  - traveling the shortest distance
  - *f* = total distance traveled

Traveling Salesperson Problem (TSP)

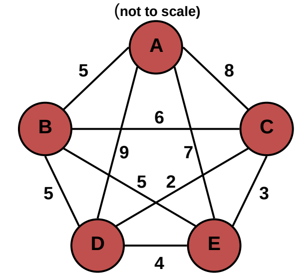
Nodes are cities

Arcs are labeled with distances between cities

Adjacency matrix (notice the graph is fully connected):

	A	B	C	D	E
A	0	5	8	9	7
B	5	0	6	5	5
C	8	6	0	2	3
D	9	5	2	0	4
E	7	5	3	4	0

5 City TSP
(not to scale)

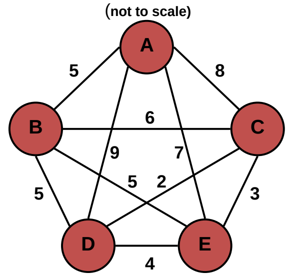


Traveling Salesperson Problem (TSP)

a solution is a permutation of cities, called a tour

	A	B	C	D	E
A	0	5	8	9	7
B	5	0	6	5	5
C	8	6	0	2	3
D	9	5	2	0	4
E	7	5	3	4	0

5 City TSP
(not to scale)



Traveling Salesperson Problem (TSP)

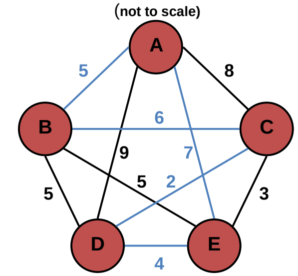
a solution is a permutation of cities, called a tour

e.g. A – B – C – D – E

assume tours return home

	A	B	C	D	E
A	0	5	8	9	7
B	5	0	6	5	5
C	8	6	0	2	3
D	9	5	2	0	4
E	7	5	3	4	0

5 City TSP
(not to scale)



Traveling Salesperson Problem (TSP)

How many solutions exist?

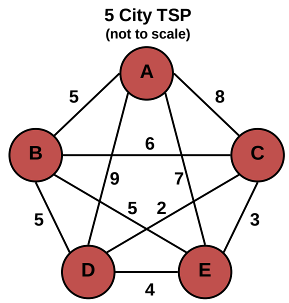
$(n-1)!/2$ where n = # of cities

n = 5 results in 12 tours

n = 10 results in 181440 tours

n = 20 results in ~$6*10^{16}$ tours

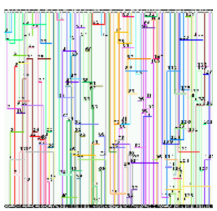
	A	B	C	D	E
A	0	5	8	9	7
B	5	0	6	5	5
C	8	6	0	2	3
D	9	5	2	0	4
E	7	5	3	4	0



Example Problems

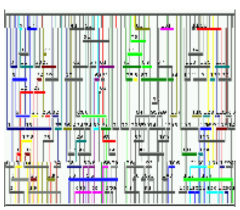
- **N-Queens**
  - Place *n* queens on *n* x *n* checkerboard so that no one can capture another
  - *f* = number of conflicting queens
- **Boolean Satisfiability**
  - Given a Boolean expression containing *n* Boolean variables, find an assignment of {T, F} to each variable so that the expression evaluates to True
  - $(A \vee \neg B \vee C) \wedge (\neg A \vee C \vee D)$
  - *f* = number of satisfied clauses

Example Problem: Chip Layout



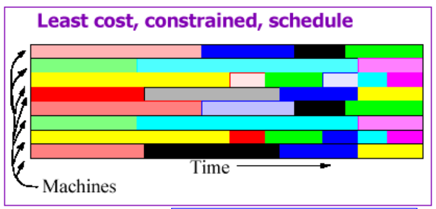
Lots of Chip Real Estate

Channel Routing



Same connectivity, much less space

Example Problem: Scheduling



Also:

parking lot layout, product design, aero-dynamic design, "Million Queens" problem, radiotherapy treatment planning, …

Local Searching

- Hard problems can be solved in a reasonable (i.e., polynomial) time by using either:
  - approximate model: find an exact solution to a simpler version of the problem
  - approximate solution: find a non-optimal solution of the original hard problem
- We'll explore means to search through a solution space by iteratively improving solutions until one is found that is optimal or near optimal

Local Searching

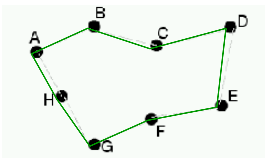
- Local searching: every node is a solution
  - operators go from one solution to another
  - can stop any time and have a valid solution
  - goal of search is to find a ***better*** solution
- No longer searching state space for a solution path and then executing the steps of the solution path
- A* isn't a local search since it searches different partial solutions by looking at the estimated cost of a *solution path*

Local Searching

- An ***operator*** is needed to transform one solution to another
- TSP: two-swap operator
  - take two cities and swap their positions in the tour
  - A-B-C-D-E with swap(A,D) yields D-B-C-A-E
  - possible since graph is fully connected
- TSP: two-interchange operator
  - reverse the path between two cities
  - A-B-C-D-E with interchange(A,D) yields D-C-B-A-E

Neighbors: TSP

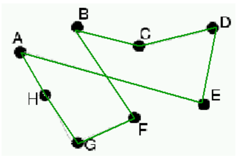
- state: A-B-C-D-E-F-G-H-A
- *f* = length of tour
- 2-interchange



A-B-C-D-E-F-G-H-A

flip

A-E-D-C-B-F-G-H-A



Local Searching

- Those solutions that can be reached with one application of an operator are in the current solution's ***neighborhood*** (“***move set***”)
- Local search considers only those solutions in the neighborhood
- The neighborhood should be much smaller than the size of the search space (otherwise the search degenerates)

Examples of Neighborhoods

- **N-queens**: Move queen in rightmost, most-conflicting column to a different position in that column
- **SAT**: Flip the assignment of one Boolean variable

Neighbors: SAT

- State: (A=T, B=F, C=T, D=T, E=T)
- *f* = number of satisfied clauses
- Neighbor: flip the assignment of one variable

(A=**F**, B=F, C=T, D=T, E=T)
(A=T, B=**T**, C=T, D=T, E=T)
(A=T, B=F, C=**F**, D=T, E=T)
(A=T, B=F, C=T, D=**F**, E=T)
(A=T, B=F, C=T, D=T, E=**F**)

$A \vee \neg B \vee C$
$\neg A \vee C \vee D$
$B \vee D \vee \neg E$
$\neg C \vee \neg D \vee \neg E$
$\neg A \vee \neg C \vee E$

Local Searching

- An evaluation function, ***f***, is used to map each solution/state to a number corresponding to the *quality* of that solution
- TSP: Use the distance of the tour path; A better solution has a shorter tour path
- Maximize *f*: called hill-climbing (gradient ascent if continuous)
- Minimize *f*: called or valley-finding (gradient descent if continuous)
- Can be used to maximize/minimize some cost

Hill-Climbing

- **Question**: What's a neighbor?
  - Problem spaces tend to have structure. A small change produces a neighboring state
  - The neighborhood must be small enough for efficiency
  - Designing the neighborhood is critical; This is the real ingenuity – not the decision to use hill-climbing
- **Question**: Pick which neighbor? The best one (greedy)
- **Question**: What if no neighbor is better than the current state? Stop

Hill-Climbing Algorithm

1. Pick initial state *s*
2. Pick *t* in neighbors(*s*) with the largest *f*(*t*)
3. **if** $f(t) \leq f(s)$ **then** stop and **return** *s*
4. *s* = *t*. **Goto** Step 2.

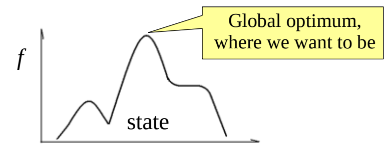
- Simple
- Greedy
- **Gets stuck at a local maximum**

Hill-Climbing (HC)

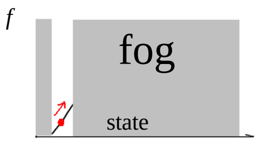
- HC *exploits* the neighborhood
  - like Greedy Best-First search, it chooses what looks best *locally*
  - but doesn't allow backtracking or jumping to an alternative path since there is no *Frontier* list
- HC is *very* space efficient
  - Like Beam search with a beam width of 1
- HC is very fast and often effective in practice

Local Optima in Hill-Climbing

- Useful mental picture: *f* is a surface ('hills') in state space



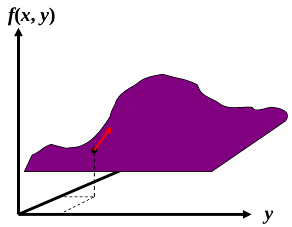
- But we can't see the entire landscape all at once. Can only see a neighborhood; like climbing in fog.



Hill-Climbing

Visualized as a 2D surface

- **Height is quality of solution**
  $f = f(x, y)$

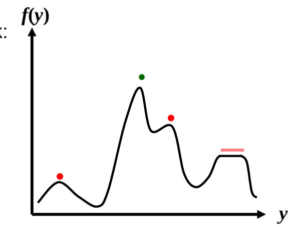


- **Solution space is a 2D surface**
- **Initial solution is a point**
- **Goal is to find a higher point on the surface of solution space**
- **Hill-Climbing follows the *direction of the steepest ascent*, i.e., where *f* increases the most**

Hill-Climbing (HC)

Solution found by HC is totally determined by the starting point;
fundamental weakness **is** getting stuck:

- **At a local maximum**
- **At plateaus and ridges**



Global maximum may *not* be found

Trade off:
greedily exploiting locality as in HC
vs. exploring state space as in BFS

Hill-Climbing with Random Restarts

- Very simple modification:
  1. When stuck, pick a random new starting state and re-run hill-climbing from there
  2. Repeat this *k* times
  3. Return the best of the *k* local optima
- Can be very effective
- Should be tried whenever hill-climbing is used
- Fast, easy to implement; works well for many applications where the solution space surface is not too "bumpy" (i.e., not too many local maxima)

Escaping Local Maxima

- HC gets stuck at a local maximum, limiting the quality of the solution found
- Two ways to modify HC:
  1. choice of neighborhood
  2. criteria for deciding to move to neighbor
- For example:
  1. choose neighbor randomly
  2. move to neighbor if it is better or, if it ***isn't***, move with some probability, p

Variations on Hill-Climbing

- **Question**: How do we make hill climbing less greedy?
  - **Stochastic hill-climbing**
    - Randomly select among better neighbors
    - The better, the more likely
    - Pros / cons compared with basic hill climbing?
- **Question**: What if the neighborhood is too large to easily compute? (e.g. N-queens if we need to pick both the column and the move within it)
  - **First-choice hill-climbing**
    - Randomly generate neighbors, one at a time
    - If better, take the move
    - Pros / cons compared with basic hill climbing?

Life Lesson #237

- **Sometimes one needs to temporarily step backward in order to move forward**
- Lesson applied to iterative, local search:
  – Sometimes one needs to move to an *inferior neighbor* in order to escape a local optimum

Hill-Climbing Example: SAT

A ∨ ¬***B*** ∨ ***C***	1
¬***A*** ∨ ***C*** ∨ ***D***	1
B ∨ ***D*** ∨ ¬***E***	0
¬***C*** ∨ ¬***D*** ∨ ¬***E***	1
¬***A*** ∨ ¬***C*** ∨ ***E***	1

Maximize:
Eval(config) =
# of satisfied clauses

Moveset:
flip any 1 variable

Example Configuration:
(1,0,1,0,1)

Variations on Hill-Climbing

WALKSAT [Selman]

- Pick a random unsatisfied clause
- Select and flip a variable from that clause:
  – With prob. *p*, pick a random variable
  – With prob. 1-*p*, pick variable that maximizes the number of satisfied clauses
- Repeat until solution found or max number of flips attempted

This is the best known algorithm for satisfying Boolean formulas

A ∨ ¬B ∨ C
¬A ∨ C ∨ D
B ∨ D ∨ ¬E
¬C ∨ ¬ D ∨ ¬ E
¬A ∨ ¬ C ∨ E

Simulated Annealing
(Stochastic Hill-Climbing)

1. Pick initial state, *s*
2. Randomly pick state *t* from neighbors of *s*
3. **if** *f*(*t*) better than *f*(*s*)
   then *s* = *t*
   else with small probability *s* = *t*
4. Goto Step 2 until bored

Simulated Annealing

Origin:

The annealing process of heated solids – Alloys manage to find a near global minimum energy state when heated and then slowly cooled

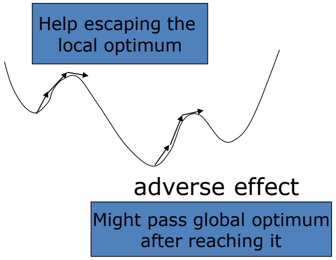
Intuition:

By allowing occasional ascent in the search process, we might be able to escape the trap of local minima

Introduced by Nicholas Metropolis in 1953

Consequences of Occasional Bad Moves

desired effect (when searching for a global min)



Idea 1: Use a small, fixed probability threshold, say, *p* = 0.1

Escaping Local Optima

- Modified HC can escape from a local optimum *but*
  - chance of making a bad move is the *same* at the beginning of the search as at the end
  - magnitude of improvement, or lack of, is ignored
- Fix by replacing fixed probability, *p*, that a bad move is accepted with a probability that ***decreases*** as the search proceeds
- Now as the search progresses, the chance of taking a bad move reduces

Control of Annealing Process

Acceptance of a search step (Metropolis Criterion) when Hill-Climbing:

- Let the performance change in the search be:
  $\Delta E = f(newNode) - f(currentNode)$
- Always accept an ascending step (i.e., better state)
  $\Delta E \geq 0$
- Accept a descending step only if it passes a test

Escaping Local Maxima

Let $\Delta E = f(newNode) - f(currentNode)$

$p = e^{\Delta E / T}$ (Boltzman's equation)

Idea: Probability decreases as neighbor gets worse

- $\Delta E \rightarrow -\infty, \; p \rightarrow 0$
  as badness of the move ***increases***
  probability of taking it ***decreases*** exponentially
- $T \rightarrow 0, \; p \rightarrow 0$
  as temperature ***decreases***
  probability of taking bad move ***decreases***

Escaping Local Maxima

Let $\Delta E = f(newNode) - f(currentNode)$

$p = e^{\Delta E / T}$ (Boltzman's equation)

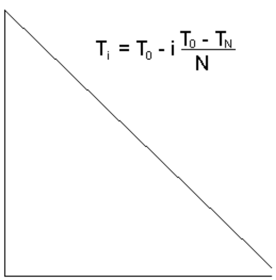
- $\Delta E << T$
  if badness of move is small compared to *T*, move is ***likely*** to be accepted
- $\Delta E >> T$
  if badness of move is large compared to *T*, move is ***unlikely*** to be accepted

Control of Annealing Process

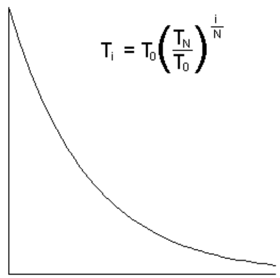
Cooling Schedule:

- *T*, the *annealing temperature*, is the parameter that control the frequency of acceptance of bad steps
- We gradually reduce temperature $T(k)$
- At each temperature, search is allowed to proceed for a certain number of steps, $L(k)$
- The choice of parameters $\{T(k), L(k)\}$ is called the ***cooling schedule***

Simple Cooling Schedules



$$T_i = T_0 - i\,\frac{T_0 - T_N}{N}$$

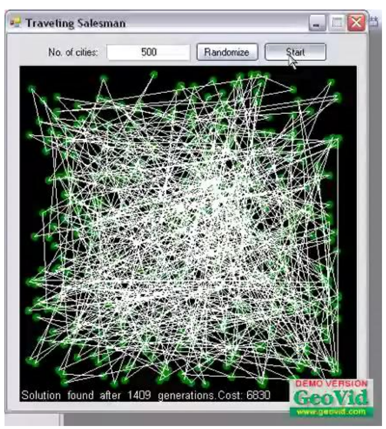


$$T_i = T_0\left(\frac{T_N}{T_0}\right)^{\frac{i}{N}}$$

Simulated Annealing
(Stochastic Hill-Climbing)

Pick initial state, *s*
k = 0
while *k* < *kmax* {
    T = temperature(*k*)
    Randomly pick state *t* from neighbors of *s*
    if *f*(*t*) > *f*(*s*) **then** *s* = *t*
    else if ($e^{(f(newNode) - f(currentNode)) / T}$) > random()
    then *s* = *t*
    k = *k* +1
    }
return *s*

SA for Solving TSP



Simulated Annealing

- Can perform multiple backward steps in a row to escape a local optimum
- Chance of finding a global optimum increased
- Fast
  - only one neighbor generated at each iteration
  - whole neighborhood isn't checked to find best neighbor as in HC
- Usually finds a good quality solution in a very short amount of time

Simulated Annealing

- Requires several parameters to be set
  - starting temperature
    - must be high enough to escape local optima
      but not too high to be random exploration of space
  - cooling schedule
    - typically exponential
  - halting temperature
- Domain knowledge helps set values:
  size of search space, bounds of maximum
  and minimum solutions

Simulated Annealing Issues

- Neighborhood design is critical. This is the real ingenuity – not the decision to use simulated annealing
- Evaluation function design often critical
- Annealing schedule often critical
- It's often cheaper to evaluate an incremental change of a previously evaluated object than to evaluate from scratch. Does simulated annealing permit that?
- What if approximate evaluation is cheaper than accurate evaluation?
- Inner-loop optimization often possible

Implementation of Simulated Annealing

- This is a stochastic algorithm; the outcome may be different at different trials
- Convergence to global optimum can only be realized in an asymptotic sense
  - With infinitely slow cooling rate, finds global optimum with probability 1

SA Discussion

- Simulated annealing is sometimes empirically much better at avoiding local maxima than hill-climbing. It is a successful, frequently-used, algorithm. Worth putting in your algorithmic toolbox.
- Sadly, not much opportunity to say anything formal about it (though there is a proof that with an infinitely slow cooling rate, you'll find the global optimum)
- There are mountains of practical, and problem-specific, papers on improvements