Informed Search

Chapter 4.1 – 4.3

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 nodes 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

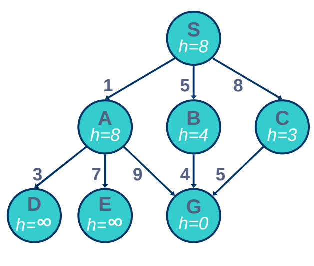
- *Use as an evaluation function, $f(n) = h(n)$, sorting nodes in the nodes list by increasing values of f*
- **Selects the node to expand that is believed to be closest (i.e., smallest f value) to a goal node**

Greedy Best-First Search

$f(n) = h(n)$

# of nodes tested: 0, expanded: 0

expnd. node	OPEN list
	{S:8}

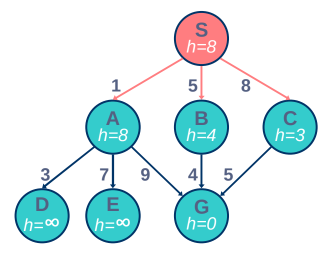


Greedy Best-First Search

$f(n) = h(n)$

# of nodes tested: 1, expanded: 1

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

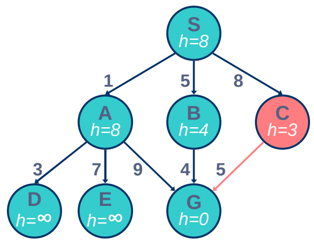


Greedy Best-First Search

f(n) = h(n)

# of nodes tested: 2, expanded: 2

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

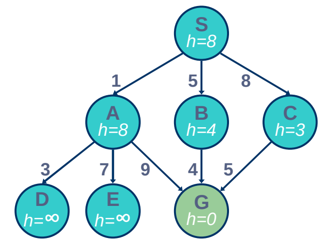


Greedy Best-First Search

f(n) = h(n)

# of nodes tested: 3, expanded: 2

expnd. node	OPEN 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

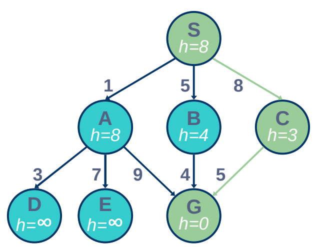


Greedy Best-First Search

f(n) = h(n)

# of nodes tested: 3, expanded: 2

expnd. node	OPEN 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***

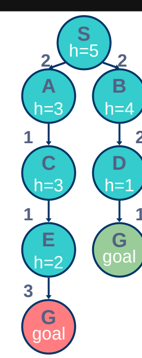
path: S,C,G
cost: 13

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 nodes 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

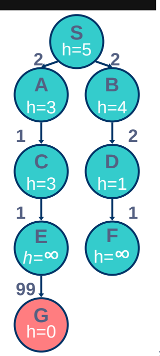
Algorithm A Search

- *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 search frontier (in nodes list) 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

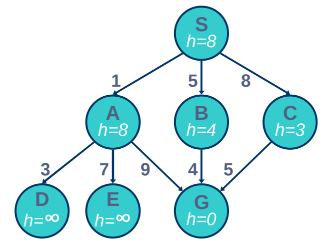
- *Use the same evaluation function used by Algorithm A, except add the constraint that for all nodes n in the search space, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost of the minimal cost path from n to a goal*
- The cost to the nearest goal is never ***overestimated***
- When $h(n) \leq h^*(n)$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

Algorithm A* Search

- **Complete**
  whenever the branching factor is finite
  (as also is required for BFS, IDS, UCS),
  and every operator has a fixed positive cost
- **Optimal/Admissible**

Example

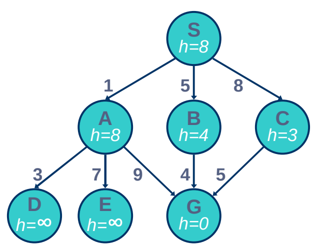
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S				
A				
B				
C				
D				
E				
G				



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

Example

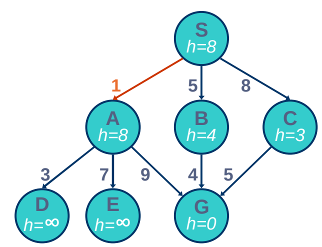
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0			
A				
B				
C				
D				
E				
G				



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

Example

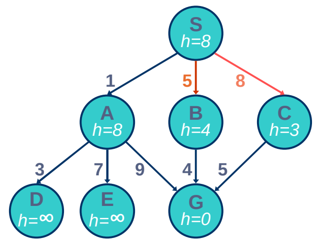
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0			
A	1			
B				
C				
D				
E				
G				



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

Example

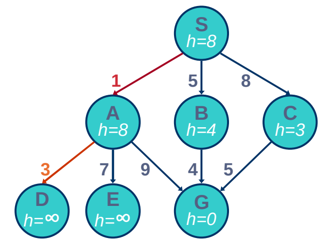
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0			
A	1			
B	5			
C	8			
D				
E				
G				



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

Example

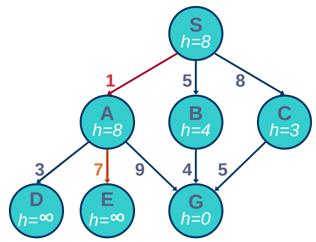
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0			
A	1			
B	5			
C	8			
D	1+3 = 4			
E				
G				



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

Example

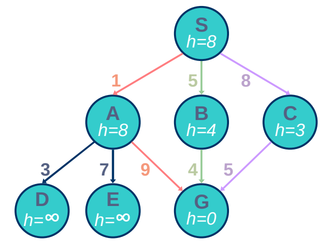
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0			
A	1			
B	5			
C	8			
D	4			
E	1+7 = 8			
G				



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

Example

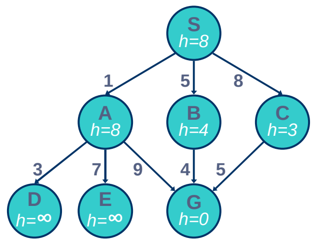
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0			
A	1			
B	5			
C	8			
D	4			
E	8			
G	10/9/13			



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

Example

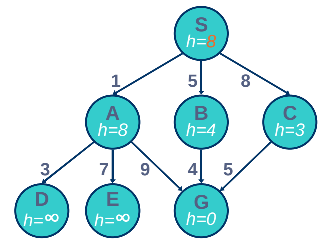
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0			
A	1			
B	5			
C	8			
D	4			
E	8			
G	10/9/13			



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

Example

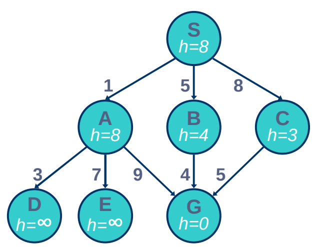
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8		
A	1	8		
B	5	4		
C	8	3		
D	4	∞		
E	8	∞		
G	10/9/13	0		



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

Example

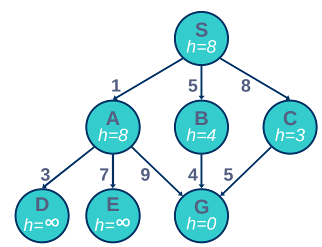
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8	8	
A	1	8	9	
B	5	4	9	
C	8	3	11	
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	



$f(n) = g(n) + h(n)$
actual cost to get from start to n
plus estimated cost from n to goal

Example

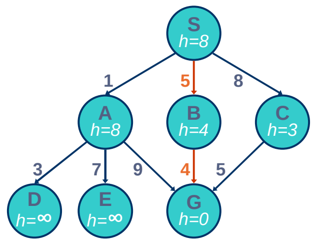
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8	8	
A	1	8	9	
B	5	4	9	
C	8	3	11	
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	



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

Example

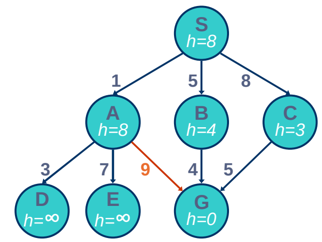
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8	8	9
A	1	8	9	
B	5	4	9	
C	8	3	11	
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	



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

Example

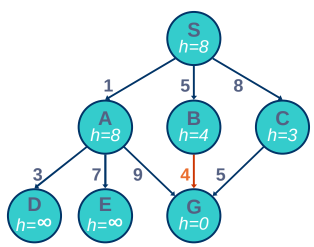
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8	8	9
A	1	8	9	9
B	5	4	9	
C	8	3	11	
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	



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

Example

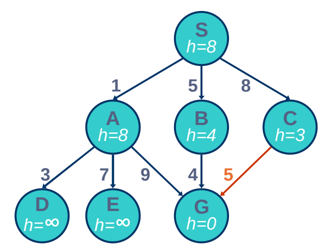
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	



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

Example

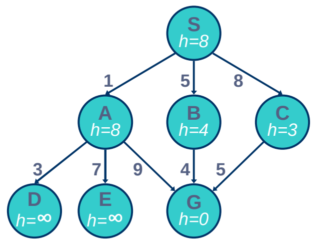
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	



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

Example

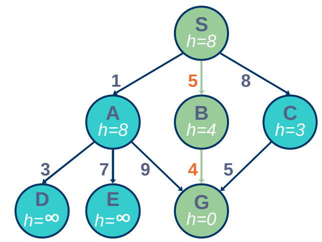
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	∞	∞	∞
E	8	∞	∞	∞
G	10/9/13	0	10/9/13	0



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

Example

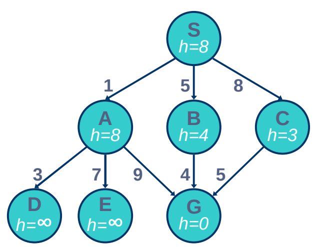
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	∞	∞	∞
E	8	∞	∞	∞
G	10/9/13	0	10/9/13	0



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

A* Algorithm

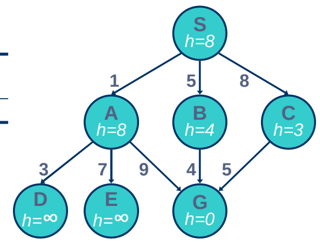
1. **Put the start node** S **in the nodes list, called** *OPEN*
2. **If** *OPEN* **is empty, exit with failure**
3. **Remove from** *OPEN* **the node** n **for which** $f(n)$ **is minimum and place it in** *CLOSED*, **the list of nodes that have been expanded**
4. **If** n **is a goal node, exit (for solution path trace back links from** n **to** S**)**
5. **Expand** n, **generating all its successors and store in them a link back to** n
   For each successor n' **of** n
   a) If n' **is not** already in either *OPEN* or *CLOSED*, calculate $h(n')$, $g(n')=g(n)+c(n,n')$, $f(n')=g(n')+h(n')$, and place n' in *OPEN*
   b) If n' **is** already in either *OPEN* or *CLOSED*, Check if $g(n')$ is lower for the newly generated n'
      i. If it **is not** lower, do nothing
      ii. If it **is** lower replace existing n' with new $g(n')$, $h(n')$, link back to this n and if in CLOSED move node from CLOSED to OPEN
6. **Goto 2**

A* Search

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

\# of nodes tested: 0, expanded: 0

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

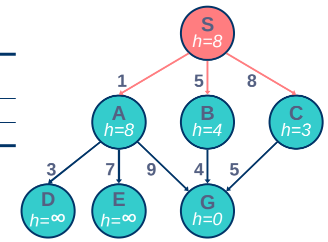


A* Search

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

\# of nodes tested: 1, expanded: 1

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

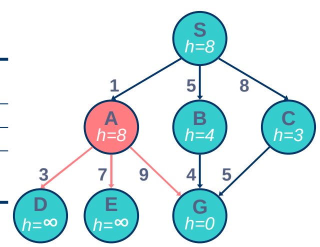


A* Search

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

\# of nodes tested: 2, expanded: 2

expnd. node	OPEN 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+∞}

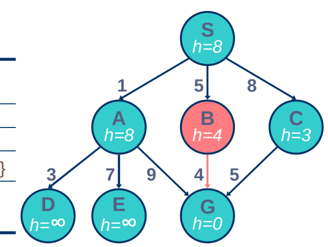


A* Search

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

\# of nodes tested: 3, expanded: 3

expnd. node	OPEN 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

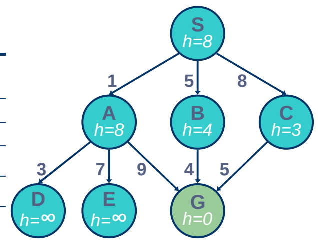


A* Search

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

# of nodes tested: 4, expanded: 3

expnd. node	OPEN 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

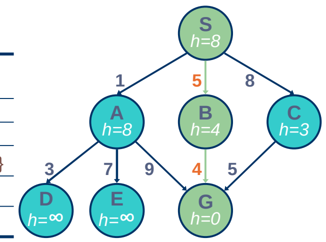


A* Search

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

# of nodes tested: 4, expanded: 3

expnd. node	OPEN 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 by Contradiction of A* Optimality

- **Let:**
  G be the optimal goal
  $G2$ be a sub-optimal goal
  f^* be the cost of the optimal path from start to G
- $g(G2) > f^*$
  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 by Contradiction of A* Optimality

- **Let n be some node on the *optimal* path but not on the path to $G2$**
- $f(n) \le f^*$
  by admissibility, since $f(n)$ never overestimates the cost to the goal it must be ≤ the cost of the optimal path
- $f(G2) \le f(n)$
  $G2$ was chosen over n for the sub-optimal goal to be found
- $f(G2) \le f^*$
  combining equations

Proof by Contradiction of A* Optimality

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

Devising Heuristics

- **Are often devised by relaxing the problem**
  compute exact cost of a solution to a *simplified* version
  - remove constraints: 8-puzzle movement
  - simplify problem: straight line distance for actual
  - see text

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:**
  using time to compute complex heuristic versus
  using time 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 $h1(n) \le h2(n) \le h^*(n)$ 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

- **Unfortunately, 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, IDS, UCS, Greedy Best-First, A, A* etc.
  - **ok** for small search spaces that are often "toy world" problems
  - **not ok** for NP-Hard problems requiring exponential time to find the optimal solution

Traveling Salesperson Problem (TSP)

- **Classic NP-Hard problem:**

  A salesman wants to visit a list of cities
  - stopping in each city only once
  - returning to the first city
  - traveling the shortest distance

Traveling Salesperson Problem (TSP)

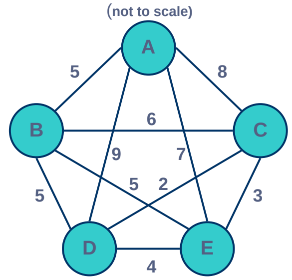
Nodes are cities

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)

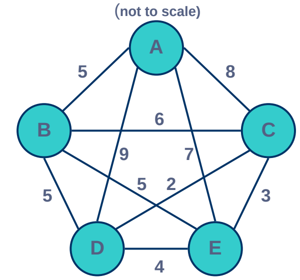


Traveling Salesperson Problem (TSP)

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)

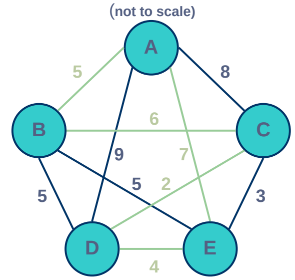
a solution is a permutation of cities, called a tour

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

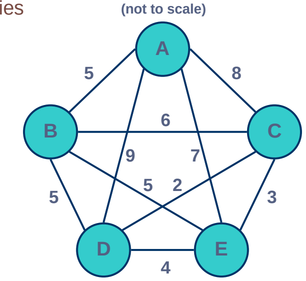
n = 5 results in 12 tours

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

5 City TSP
(not to scale)



Other Example Problems

- **N-Queens**
  - Place N queens on N x N checkerboard so that no one can capture another
- **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 T
  - $(A \vee \neg B \vee C) \wedge (\neg A \vee C \vee D)$

Local Searching

- **Hard problems can be solved in a reasonable time (i.e., polynomial) 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
- **Next 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
  - search is now finding a *better* solution
- ∗ ***No longer searching state space for a solution path and then executing the steps of the solution path***
- **A* isn't local searching since it searches different partial solutions by looking at estimated cost of a *solution path***

Local Searching

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

Local Searching

- **Those solutions that can be reached with one application of an operator are in the current solution's neighborhood**
- **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)**

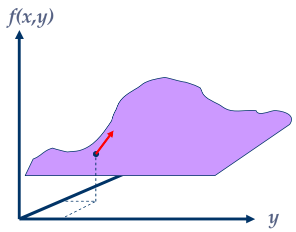
Local Searching

- ∗ ***An evaluation function $f(n)$ is used to map each solution 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 $f(n)$:**
  called hill climbing (gradient ascent if continuous)
- **Minimize $f(n)$:**
  called or valley finding (gradient descent if continuous)
- **Can be used to maximize/minimize some cost**

Local Searching

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**



Local Searching

```
Node hillClimbing (Problem problem) {
  Node currentNode = new Node(problem.getStartPoint());
  while (true) {
    Node newNode = best neighbor of currentNode;
    //neighbor with highest evaluation (lowest for VF)
    if (problem.eval(newNode) >= problem.eval(currentNode))
      //(<= for VF: valley finding)
      //to avoid cycles a CLOSED list could be used
      currentNode = newNode;
    else
      return currentNode;
  }
}
```


Hill-Climbing (HC)

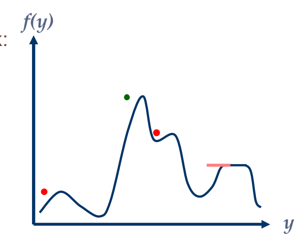
- **HC *exploits* the neighborhood**
  - like Greedy Best-First search it chooses what looks best *locally*
  - but doesn't allow backtracking or jumping to an alternative path since there is no OPEN list
- **HC is *very* space efficient**
  - Like Beam search with a beam width of 1 recall beam width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

Hill-Climbing (HC)

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

- **at local maximum**
- **at plateaus and ridges**
- **global maximum may *not* be found**

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



Hill-Climbing with Restarts

- **Run HC multiple times (independently) with randomly generated start solutions**
- **Use best solution found**
- **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 neighbor
  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, accept with some fixed probability p

Escaping Local Maxima

- **Modified HC can escape from local maxima 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 chances of taking a bad move reduces**

Escaping Local Maxima

- **Lets change to Valley Finding (VF):**
  - want to find the global minima (lowest solution)
  - search for a new node with a lower value of $f(n)$
  - replace fixed probability p with a temperature T that decreases as the search proceeds:

$$e^{(eval(currentNode) - eval(newNode) / T}$$

Escaping Local Maxima

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

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

- $\Delta E \to -\infty, p \to 0$

  as badness (VF) of the move *increases*
  probability of taking it *decreases* exponentially
- $T \to 0, p \to 0$

  as temperature *decreases*
  probability of taking bad move *decreases*

Escaping Local Maxima

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

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

- $\Delta E << T$

  if badness of move (VF) small compare to T
  move likely to be accepted
- $\Delta E >> T$

  if badness of move (VF) large compare to T
  move unlikely to be accepted

Simulated Annealing (SA)

```
//finds minima, VF valley finding
Node simulatedAnnealing (Problem problem) {
  int  temperature = problem.getIntialTemperature();
  Node currentNode = new Node(problem.getStartPoint());
  Node bestNode    = currentNode;
  while (temperature > problem.getStoppingTemperature()) {
    Node newNode = random neighbor of currentNode;
    check if newNode is accepted
    check if newNode is better than bestNode
    temperature  = problem.schedule(temperature);
  }
  return bestNode;
}
```


Simulated Annealing (SA)

```
//check if newNode is accepted
int deltaE = problem.eval(currentNode) - problem.eval(newNode));
if (deltaE > 0) //VF: newNode is better
  currentNode = newNode;
else //allow for backtracking
  currentNode = newNode with probability p = e^(deltaE/T);

//check if newNode is better than bestNode
//needed since current node may not be best
if (problem.eval(currentNode) < problem.eval(bestNode))
  bestNode = currentNode;
```

Simulated Annealing (SA)

- **Can perform multiple backward steps in a row to escape local optimum**
- **Chance of finding a global optima increased**
- **Fast**
  - only one neighbor generated 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**

77

Simulated Annealing (SA)

- **Requires 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**

78

Summary

- Systematic Searching
  - ∗ *look through state space for a goal from which solution path can be determined*
  - **nodes**: state descriptions, partial solution path
  - **arcs**: operator changes state for some cost
  - **solution**: sequence of operators that change from start to goal state

79

Summary

- Systematic Searching
  - Uninformed: domain info not used to guide search
    - complete/optimal if arc costs uniform: BFS, IDS
    - complete/optimal: UCS (uses arc costs)
    - not complete/optimal: DFS
  - Informed: domain info weakly used to guide search
    - $g(n)$: cost from start to n
    - $h(n)$: estimated cost from n to goal, heuristic
    - $f(n) = g(n) + h(n)$: estimated cost of solution through n
    - complete/optimal: A* when $h(n)$ admissible
    - not complete/optimal: Greedy Best-First, Beam, A

80

Summary

- **Local Searching**
  * *Iteratively improve solution*

  - **nodes**: complete solution
  - **arcs**: operator changes to another solution
  - can stop at any time
  - technique suited for:
    - hard problems, e.g., TSP
    - optimization problems

Summary

- **Local Searching**
  - $f(n)$ evaluates quality of solution by weakly using domain knowledge
  - HC: maximizes $f(n)$, VF: minimizes $f(n)$
    - solution found determined by starting point
    - can get stuck, which prevents finding global optimum
  - SA: explores, then settles down
    - bad moves accepted with probability that decreases as the search progress (T decreases) with the badness of move (ΔE worsens)
    - requires parameters to be set