Module 2

Problem Solving using Search- (Single agent search)

Version 1 CSE IIT, Kharagpur

Lesson 6

Informed Search Strategies-II

Version 1 CSE IIT, Kharagpur

3.3 Iterative-Deepening A*

3.3.1 IDA* Algorithm

Iterative deepening A* or IDA* is similar to iterative-deepening depth-first, but with the following modifications:
The depth bound modified to be an f-limit

1. Start with limit = h(start)
2. Prune any node if f(node) > f-limit
3. Next f-limit=minimum cost of any node pruned

The cut-off for nodes expanded in an iteration is decided by the f-value of the nodes.

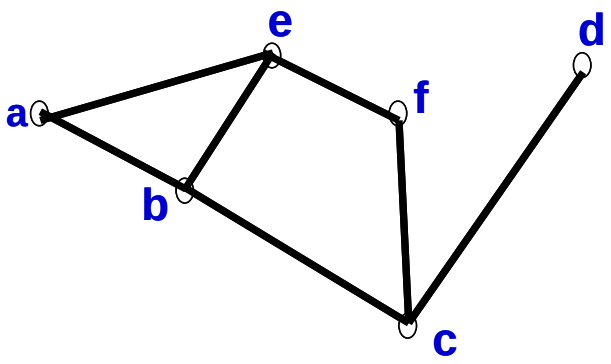


Figure 1

Consider the graph in Figure 3. In the first iteration, only node a is expanded. When a is expanded b and e are generated. The f value of both are found to be 15.

For the next iteration, a f-limit of 15 is selected, and in this iteration, a, b and c are expanded. This is illustrated in Figure 4.

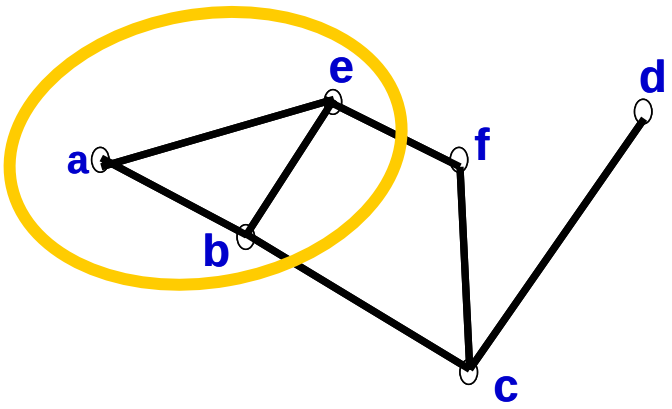


Figure 2: f-limit = 15

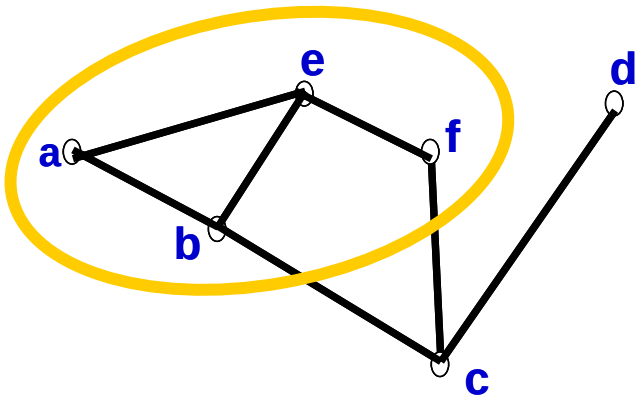


Figure 3: f-limit = 21

3.3.2 IDA* Analysis

IDA* is complete & optimal Space usage is linear in the depth of solution. Each iteration is depth first search, and thus it does not require a priority queue.

The number of nodes expanded relative to A* depends on # unique values of heuristic function. The number of iterations is equal tit h number of distinct f values less than or equal to C*.

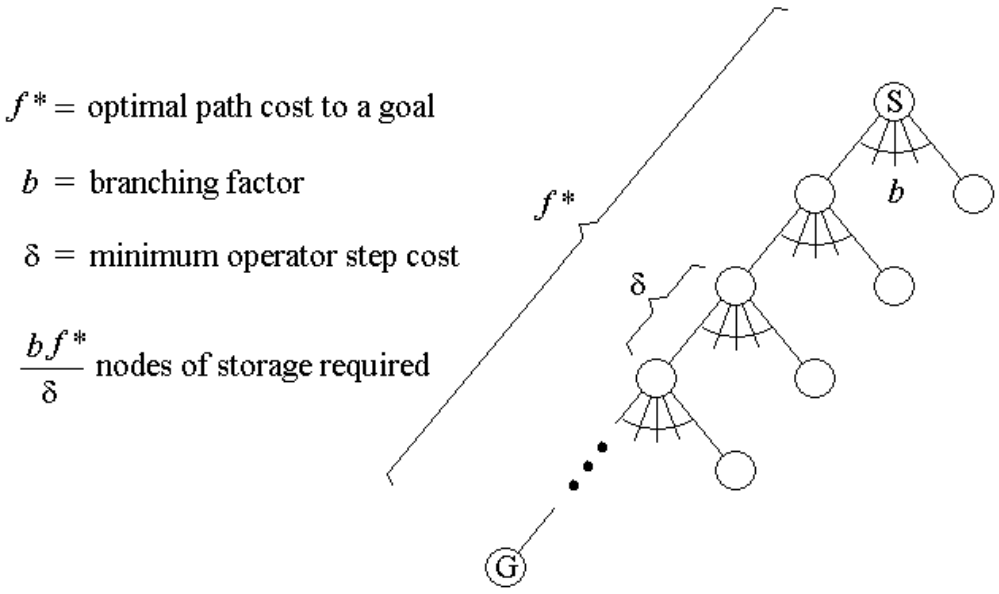
- In problems like 8 puzzle using the Manhattan distance heuristic, there are few possible f values (f values are only integral in this case.). Therefore the number of node expansions in this case is close to the number of nodes A* expands.
- But in problems like traveling salesman (TSP) using real valued costs, : each f value may be unique, and many more nodes may need to be expanded. In the worst case, if all f values are distinct, the algorithm will expand only one new node per iteration, and thus if A* expands N nodes, the maximum number of nodes expanded by IDA* is 1+2+…+ N = O(N2)

Why do we use IDA*? In the case of A*, it I usually the case that for slightly larger problems, the algorithm runs out of main memory much earlier than the algorithm runs out of time. IDA* can be used in such cases as the space requirement is linear. In fact 15-puzzle problems can be easily solved by IDA*, and may run out of space on A*.

IDA* is not thus suitable for TSP type of problems. Also IDA* generates duplicate nodes in cyclic graphs. Depth first search strategies are not very suitable for graphs containing too many cycles.

Space required : O(bd)

IDA* is complete, optimal, and optimally efficient (assuming a consistent, admissible heuristic), and requires only a polynomial amount of storage in the worst case:



3.4 Other Memory limited heuristic search

IDA* uses very little memory
Other algorithms may use more memory for more efficient search.

3.4.1 RBFS: Recursive Breadth First Search

RBFS uses only linear space.
It mimics best first search.
It keeps track of the f-value of the best alternative path available from any ancestor of the current node.
If the current node exceeds this limit, the alternative path is explored.
RBFS remembers the f-value of the best leaf in the forgotten sub-tree.

```
RBFS (node: N, value: F(N), bound: B)

   IF f(N)>B, RETURN f(N)
   IF N is a goal, EXIT algorithm
   IF N has no children, RETURN infinity
   FOR each child Ni of N,
         IF f(N)<F(N), F[i] := MAX(F(N),f(Ni))
         ELSE F[i] := f(Ni)
sort Ni and F[i] in increasing order of F[i]
   IF only one child, F[2] := infinity
   WHILE (F[1] <= B and F[1] < infinity)
            F[1] := RBFS(N1, F[1], MIN(B, F[2]))
     insert Ni and F[1] in sorted order
RETURN F[1]
```

3.4.2 MA* and SMA*

MA* and SMA* are restricted memory best first search algorithms that utilize all the memory available.
The algorithm executes best first search while memory is available.
When the memory is full the worst node is dropped but the value of the forgotten node is backed up at the parent.

3.5 Local Search

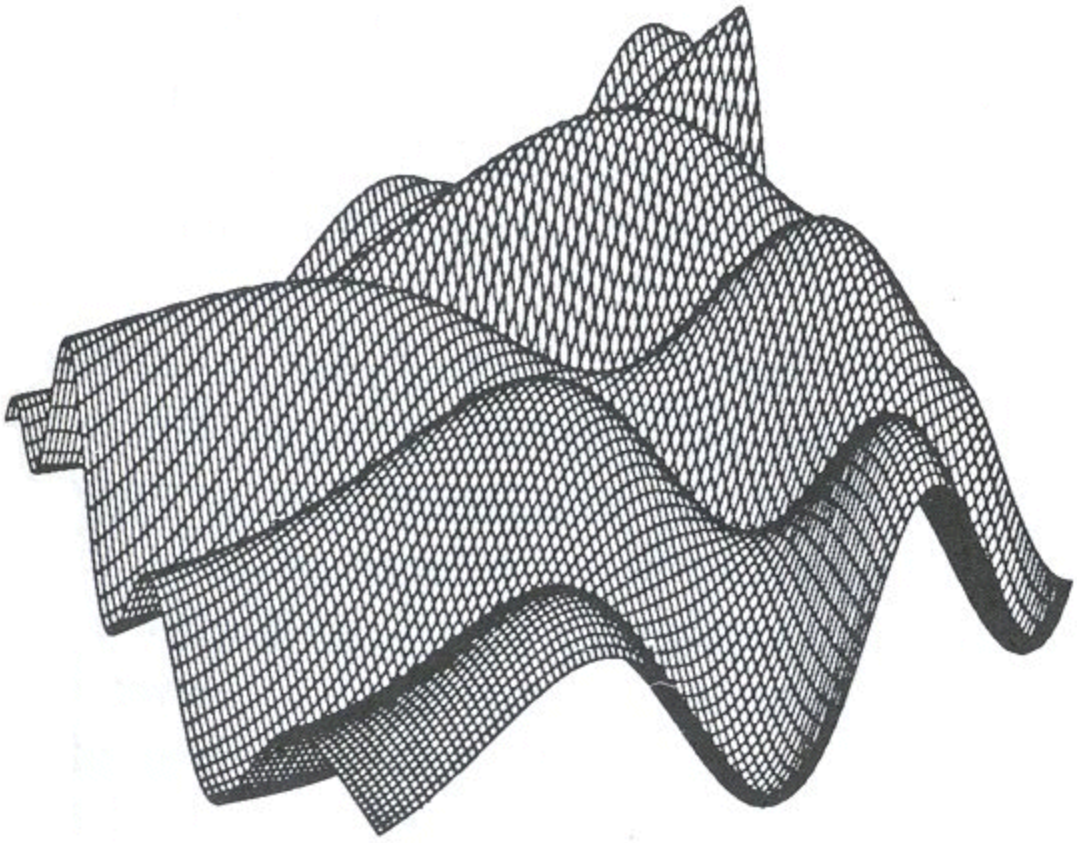
Local search methods work on complete state formulations. They keep only a small number of nodes in memory.
Local search is useful for solving optimization problems:

- Often it is easy to find a solution
- But hard to find the best solution

Algorithm goal:
find optimal configuration (e.g., TSP),

- Hill climbing
- Gradient descent
- Simulated annealing
- For some problems the state description contains all of the information relevant for a solution. Path to the solution is unimportant.
- Examples:
  - map coloring
  - 8-queens
  - cryptarithmetic

- Start with a state configuration that violates some of the constraints for being a solution, and make gradual modifications to eliminate the violations.
- One way to visualize iterative improvement algorithms is to imagine every possible state laid out on a landscape with the height of each state corresponding to its goodness. Optimal solutions will appear as the highest points. Iterative improvement works by moving around on the landscape seeking out the peaks by looking only at the local vicinity.



3.5.1 Iterative improvement

In many optimization problems, the path is irrelevant; the goal state itself is the solution. An example of such problem is to find configurations satisfying constraints (e.g., n-queens).

Algorithm:

- Start with a solution
- Improve it towards a good solution

3.5.1.1 Example:

N queens
Goal: Put n chess-game queens on an n x n board, with no two queens on the same row, column, or diagonal.

Example:

Chess board reconfigurations

Here, goal state is initially unknown but is specified by constraints that it must satisfy

Hill climbing (or gradient ascent/descent)

Iteratively maximize "value" of current state, by replacing it by successor state that has highest value, as long as possible.

Note: minimizing a "value" function v(n) is equivalent to maximizing –v(n),

thus both notions are used interchangeably.

Hill climbing – example

Complete state formulation for 8 queens
Successor function: move a single queen to another square in the same column
Cost: number of pairs that are attacking each other.
Minimization problem

Hill climbing (or gradient ascent/descent)

- *Iteratively maximize "value" of current state, by replacing it by successor state that has highest value, as long as possible.*

Note: minimizing a "value" function v(n) is equivalent to maximizing –v(n), thus both notions are used interchangeably.

- Algorithm:
    1. determine successors of current state
    2. choose successor of maximum goodness (break ties randomly)
    3. if goodness of best successor is less than current state's goodness, stop
    4. otherwise make best successor the current state and go to step 1
- No search tree is maintained, only the current state.
- Like greedy search, but only states directly reachable from the current state are considered.
- Problems:

Local maxima
Once the top of a hill is reached the algorithm will halt since every possible step leads down.

Plateaux
If the landscape is flat, meaning many states have the same goodness, algorithm degenerates to a random walk.

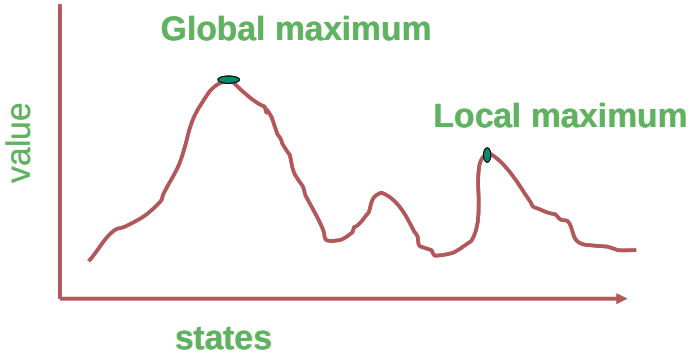
Ridges
If the landscape contains ridges, local improvements may follow a zigzag path up the ridge, slowing down the search.

- Shape of state space landscape strongly influences the success of the search process. A very spiky surface which is flat in between the spikes will be very difficult to solve.
- Can be combined with nondeterministic search to recover from local maxima.
- **Random-restart hill-climbing** is a variant in which reaching a local maximum causes the current state to be saved and the search restarted from a random point. After several restarts, return the best state found. With enough restarts, this method will find the optimal solution.
- **Gradient descent** is an inverted version of hill-climbing in which better states are represented by lower *cost* values. Local *minima* cause problems instead of local maxima.

Hill climbing - example

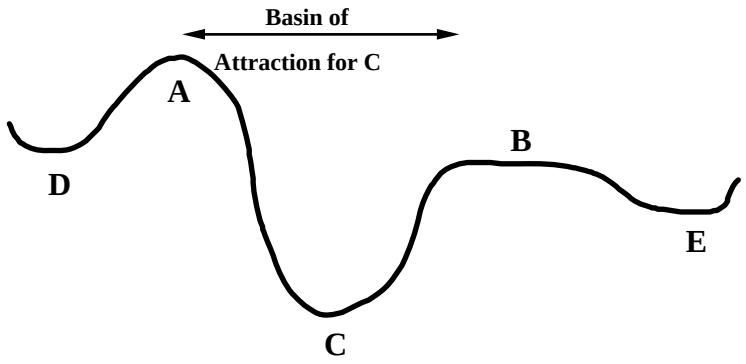
- *Complete state formulation for 8 queens*
  - Successor function: move a single queen to another square in the same column
  - Cost: number of pairs that are attacking each other.
- *Minimization problem*

- *Problem: depending on initial state, may get stuck in local extremum.*



Minimizing energy

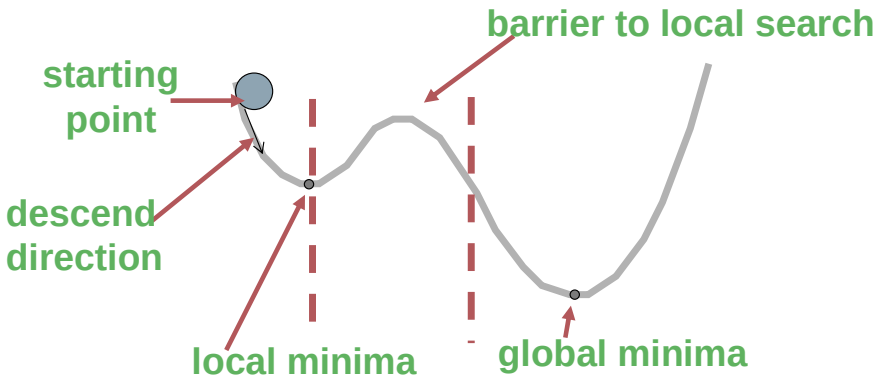
- Compare our state space to that of a physical system that is subject to natural interactions
- Compare our value function to the overall potential energy E of the system.
- On every updating, we have DE ≤ 0



Hence the dynamics of the system tend to move E toward a minimum.

We stress that there may be different such states — they are *local* minima. Global minimization is not guaranteed.

- *Question: How do you avoid this local minima?*



Consequences of Occasional Ascents

Simulated annealing: basic idea

- *From current state, pick a random successor state;*
- *If it has better value than current state, then "accept the transition," that is, use successor state as current state;*

Simulated annealing: basic idea

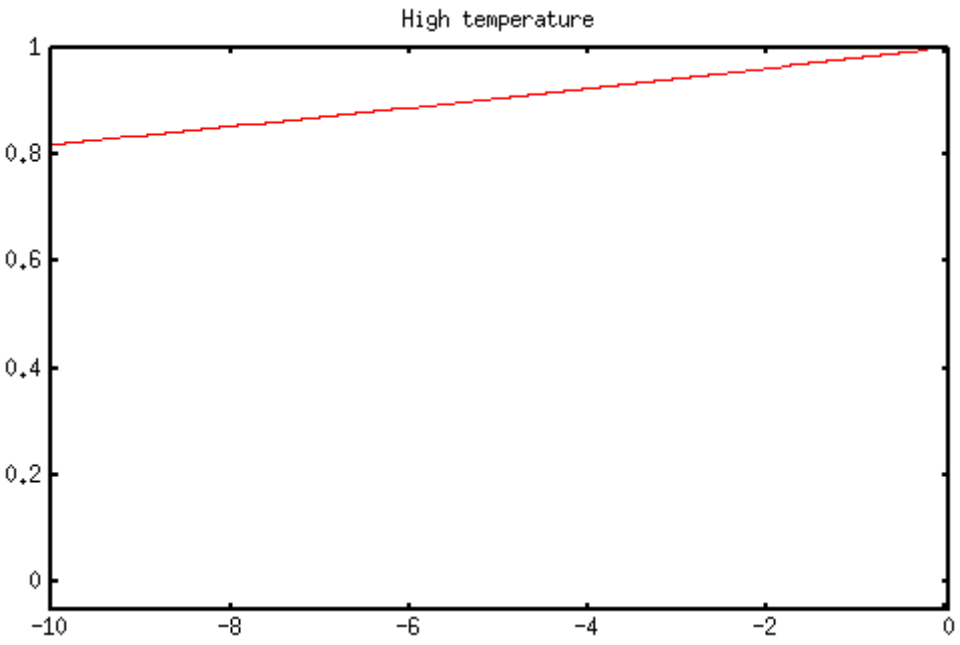
- *Otherwise, do not give up, but instead flip a coin and accept the transition with a given probability (that is lower as the successor is worse).*
- *So we accept to sometimes "un-optimize" the value function a little with a non-zero probability.*

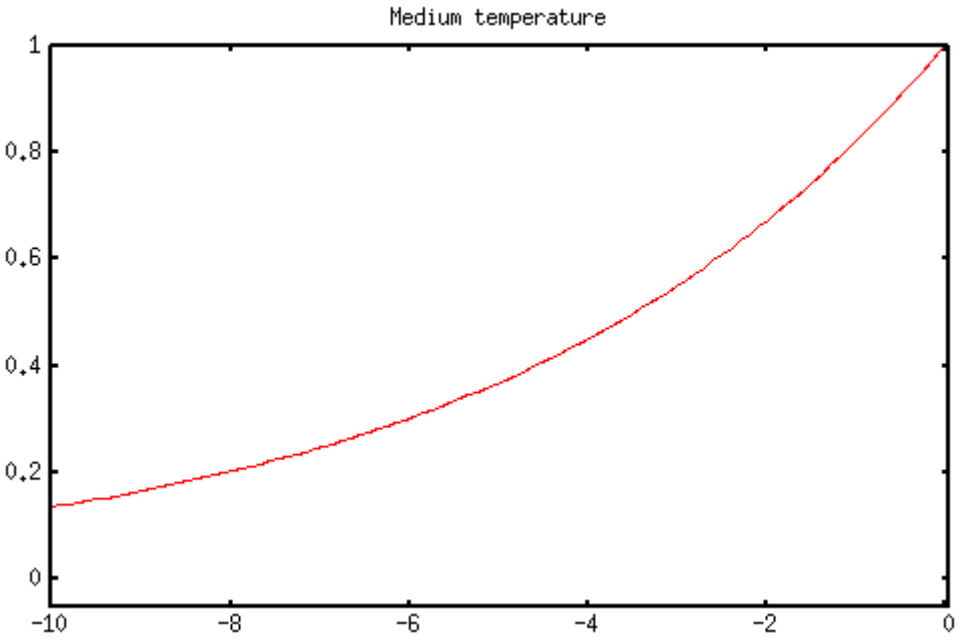
- Instead of restarting from a random point, we can allow the search to take some downhill steps to try to escape local maxima.
- Probability of downward steps is controlled by **temperature** parameter.
- High temperature implies high chance of trying locally "bad" moves, allowing nondeterministic exploration.
- Low temperature makes search more deterministic (like hill-climbing).
- Temperature begins high and gradually decreases according to a predetermined **annealing schedule**.
- Initially we are willing to try out lots of possible paths, but over time we gradually settle in on the most promising path.
- If temperature is lowered slowly enough, an optimal solution will be found.
- In practice, this schedule is often too slow and we have to accept suboptimal solutions.

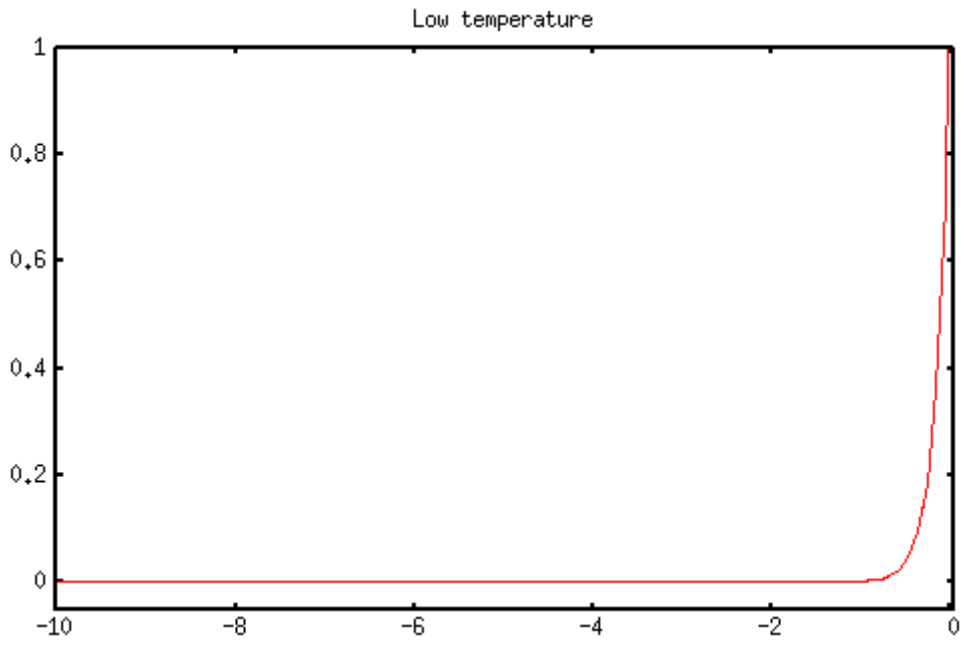
Algorithm:

```
set current to start state
for time = 1 to infinity {
   set Temperature to annealing_schedule[time]
   if Temperature = 0 {
      return current
   }
   randomly pick a next state from successors of current
   set ΔE to value(next) - value(current)
   if ΔE > 0 {
      set current to next
   } else {
      set current to next with probability e^(ΔE/Temperature)
   }
}
```

- Probability of moving downhill for negative ΔE values at different temperature ranges:


High temperature
1
0.8
0.6
0.4
0.2
0
-10
-8
-6
-4
-2
0


Medium temperature
1
0.8
0.6
0.4
0.2
0
-10
-8
-6
-4
-2
0



Other local search methods

- *Genetic Algorithms*

Questions for Lecture 6

1. Compare IDA* with A* in terms of time and space complexity.
2. Is hill climbing guaranteed to find a solution to the n-queens problem ?
3. Is simulated annealing guaranteed to find the optimum solution of an optimization problem like TSP ?

1. Suppose you have the following search space:

State	next	cost
A	B	4
A	C	1
B	D	3
B	E	8
C	C	0
C	D	2
C	F	6
D	C	2
D	E	4
E	G	2
F	G	8

a. Draw the state space of this problem.
b. Assume that the initial state is **A** and the goal state is **G**. Show how each of the following search strategies would create a search tree to find a path from the initial state to the goal state:

    i. Uniform cost search
    ii. Greedy search
    iii. A* search

At each step of the search algorithm, show which node is being expanded, and the content of *fringe*. Also report the eventual solution found by each algorithm, and the solution cost.