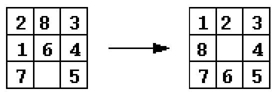
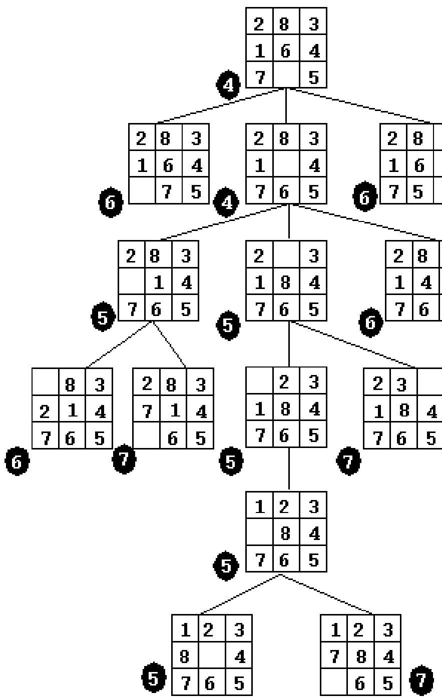
# A* Search for the 8-puzzle problem



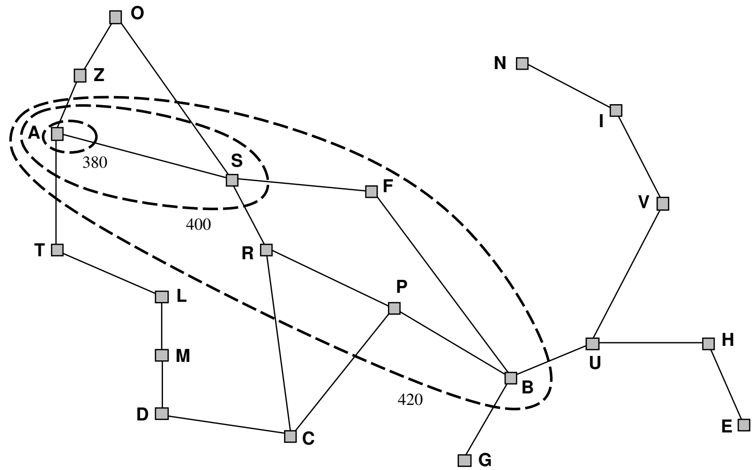
```
Evaluation function: f(n) = g(n) + W(n)
where
  g(n)- length of the path from the start
        node to n, and
  W(n)- number of misplaced tiles
```





# Behavior of A*

- Choose an h function that never overestimates the cost to reach the goal. Such an h is called an admissible heuristic.
- Let $f^*$ be cost of the optimal solution path.
  - A* expands all nodes with $f(n) < f^*$.
  - A* expands some nodes with $f(n) = f^*$ before finding goal.



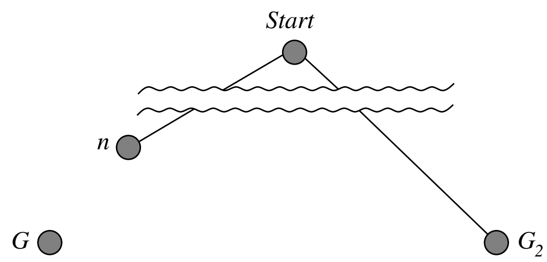
# Completeness of A*

- A* is complete if branching factor is finite and costs of actions are no smaller than some constant $\delta > 0$.
- **Proof**: If the two conditions are met, the number of nodes with $f(n) < f^*$ is finite.



# Optimality of A*

- $h$ is *admissible* if $h(n)$ is never larger than the actual cost.
- If $h$ is admissible, then A* finds an optimal goal.
- **Proof**: $G$-optimal, $G_2$-suboptimal.
It is impossible for A* to find $G_2$ before $G$.



# Complexity of A*

- Complexity:
  - Number of nodes expanded is exponential in the length of the solution.
  - All generated nodes are kept in memory.
  - With a good heuristic, significant savings are still possible compared to uninformed search methods.
  - Admissible heuristic functions that give higher values tend to make the search more efficient.
- Memory-bounded extensions to A*:
  - Iterative deepening A* (IDA*)
  - Simplified memory-bounded A* (SMA*)



# Iterative Deepening A* (IDA* Algorithm

- Each iteration is a depth-first search, similar to Iterative Deepening.
- The depth-first search is modified to use an f-cost limit rather than a depth limit.
- Each iteration expands all nodes inside the coutour for the current f-limit.

```
function IDA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: f-limit, the current f-COST limit
          root, a node

  root ← MAKE-NODE(INITIAL-STATE[problem])
  f-limit ← f-COST(root)
  loop do
     solution, f-limit ← DFS-CONTOUR(root, f-limit)
     if solution is non-null then return solution
     if f-limit = ∞ then return failure; end
```

```
function DFS-CONTOUR(node, f-limit) returns a solution sequence and a new f-COST limit
  inputs: node, a node
          f-limit, the current f-COST limit
  static: next-f, the f-COST limit for the next contour, initially ∞

  if f-COST[node] > f-limit then return null, f-COST[node]
  if GOAL-TEST[problem](STATE[node]) then return node, f-limit
  for each node s in SUCCESSORS(node) do
     solution, new-f ← DFS-CONTOUR(s, f-limit)
     if solution is non-null then return solution, f-limit
     next-f ← MIN(next-f, new-f); end
  return null, next-f
```

## Heuristic Functions for 8-Puzzle

- Some possible candidates (admissible heuristic functions):
  - Number of tiles that are in the wrong position ($h_1$)
  - Sum of the city block distances of the tiles from their goal positions ($h_2$)
- Comparison of iterative deepening search, A* with $h_1$, and A* with $h_2$:

| | Search Cost | | | Effective Branching Factor | | |
|---|---|---|---|---|---|---|
| $d$ | IDS | A*($h_1$) | A*($h_2$) | IDS | A*($h_1$) | A*($h_2$) |
| 2 | 10 | 6 | 6 | 2.45 | 1.79 | 1.79 |
| 4 | 112 | 13 | 12 | 2.87 | 1.48 | 1.45 |
| 6 | 680 | 20 | 18 | 2.73 | 1.34 | 1.30 |
| 8 | 6384 | 39 | 25 | 2.80 | 1.33 | 1.24 |
| 10 | 47127 | 93 | 39 | 2.79 | 1.38 | 1.22 |
| 12 | 364404 | 227 | 73 | 2.78 | 1.42 | 1.24 |
| 14 | 3473941 | 539 | 113 | 2.83 | 1.44 | 1.23 |
| 16 | – | 1301 | 211 | – | 1.45 | 1.25 |
| 18 | – | 3056 | 363 | – | 1.46 | 1.26 |
| 20 | – | 7276 | 676 | – | 1.47 | 1.27 |
| 22 | – | 18094 | 1219 | – | 1.48 | 1.28 |
| 24 | – | 39135 | 1641 | – | 1.48 | 1.26 |



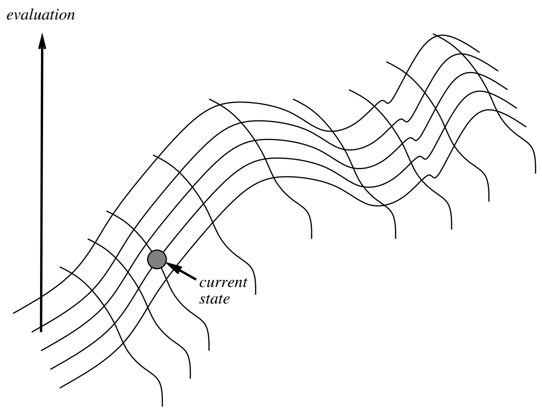
## Iterative Improvement Algorithms

- Problems discussed so far — solutions consist of sequence of actions.
- Now funciton optimization problems:

$$\arg\max_x f(x)$$

- Examples: 8-queens, VLSI design.

# Iterative Improvement Algorithms (cont'd)




- Usually keep track of only the current state, without looking ahead beyond the immediate neighbors.
- Two major classes:
  - Hill-climbing or gradient descent algorithms:
    - deterministic
    - always try to make changes that improve the current state
    - can get stuck in locally optimal solutions.
  - Simulated annealing algorithms:
    - probabilistic
    - sometimes make changes that make things worse temporarily
    - try to escape from locally optimal solutions.



# Hill-Climbing Search

```
function HILL-CLIMBING(problem) returns a solution state
   inputs: problem, a problem
   static: current, a node
           next, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      next ← a highest-valued successor of current
      if VALUE[next] < VALUE[current] then return current
      current ← next
   end
```

# Simulated Annealing Algorithm

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  static: current, a node
          next, a node
          T, a "temperature" controlling the probability of downward steps

  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
     T ← schedule[t]
     if T=0 then return current
     next ← a randomly selected successor of current
     ΔE ← VALUE[next] – VALUE[current]
     if ΔE > 0 then current ← next
     else current ← next only with probability e^(ΔE/T)
```