Artificial Intelligence

Augustin Cosse.

Fall 2020

September 14, 2020

Knowledge Based agents (Recap I)

- Agent = something that perceives and acts in a given environment.
- An Ideal Agent is an agent that always chooses the action that is expected to maximize its performance measure
- An agent is autonomous when its actions only depends on its own experience (i.e the experience that it has accumulated through time) and not on any knowledge that was built in by the programmer.

Knowledge Based agents (Recap II)

- Different types of agents are usually classified based on how they take their decisions as well as on the information they use in the decision process. The design of an agent depend on 4 main aspects:
  - Percepts
  - Actions
  - Goals
  - Environment
- Reflex agents respond immediately to percepts. Goal-based agents act in order to maximize their goals, and utility based agents try to maximize their level of happiness.

Environment

- There are several ways to program an environment. We have already taken one approach through object oriented programming in python.
- In general, when designing our environment, we will often want to

Implementing the environment (I)

input : `state` (initial state of the env.)
`Update()` or `Step()` (to update the environment)
`agents` (a set of agents)
`termination` (stop criterion for iterations)

while `termination` *is not true* **do**
  for *each* `agent` *in* `agents` **do**
    `percept[agent]` $\leftarrow$ `agent.getPercept(state)`
  end
  for *each* `agent` *in* `agents` **do**
    `action[agent]` $\leftarrow$
     `agent.program(agent.Percept(state))`
  end
  `state` $\leftarrow$ `UpdateFun(action, agents,state)`
end

Algorithm 1: Implementation of an Environment

input : `state` (initial state of the env.)
`Update()` or `Step()`
`agents` (a set of agents)
`termination` (stop criterion for iterations)
loc. var.: `scores` (vector of size `size[agents]`)
while `termination` *is not true* **do**
  for *each* `agent` *in* `agents` **do**
    `percept[agent]` ← `agent.getPercept(state)`
  end
  for *each* `agent` *in* `agents` **do**
    `action[agent]` ←
     `agent.program(agent.Percept(state))`
  end
  `state` ← `UpdateFun(action, agents,state)`
  `scores` ← `PerformanceFun(scores, agents,state)`
end
return : `scores`

- Ideally we should define our agent to work for a given family of different environments. Using object oriented programming is thus a good idea in this case.
- We could for example design a chess program to compete against a set of different human or machine opponents.
- When evaluating the performance of the agent, we will then select particular instances of environments at random, measure the performance of our agent on those different environments and then compute the average of the perfromances over each of the agents
- Finally note that when designing the agent class and the environment class we will want to be careful about the disctinction between the agent's state variable and the environment state's variable.

Solving Problems by searching

- Now that we have discussed the main families of agents, we will consider how an agent can act by establishing golas and then consider particular sequences of actions that might achieve those goals.
- A goal together with a set of means to achieve this goals is called a problem.
- The process of exploring what the means can do is called a search

Solving Problems by searching

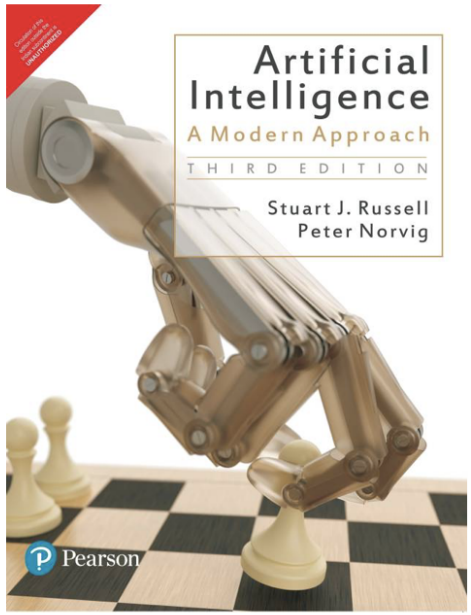
- Simple reflex agents are unable to plan ahead
- Such agents are limited in what they can achieve becaue their actions are detemrined only by the current percept
- Furthermore, they have no knowledge of what their actions do or what they are trying to achieve.

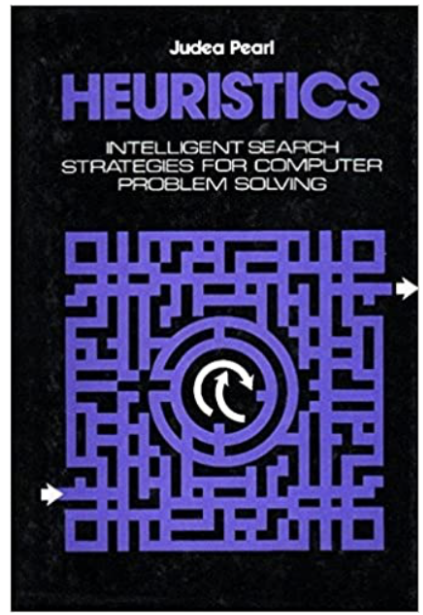
Solving Problems by searching

- Given a precise definition of a problem is relatively straighforward to construct a search process for finding solutions.
- We make the distinction between Blind or Uninformed Search Methods and Informed Search methods:
  - In Uninformed Search, the order in which the nodes are expanded depends only on information gathered by the search but is unaffected by the character of the unexplored portion of the graph, not even the goal criterion
  - Informed Search Methods on the contrary use partial information about the problem domain and about the nature of the goal to help guide the search toward the more promising solutions

Solving Problems by searching

- Good additional references for this week :
  - Pearl, *Heuristics, Intelligent Search Strategies for Computer Probem Solving*, Chap.
  - Russel and Norvig, *Artificial Intelligence, A modern Approach*, Chap. 1-3.





Search Space and Problem representations

- Most problems can be posed either as Optimization tasks (e.g. Road map, travelling salesman,..) or Satisfaction tasks (e.g 8 queens, counterfeit problem)
  - In Optimization problems, the objective is not just to exhibit a formal object satisfying an established set of criteria but also to ascertain that this object possesses qualities unmatched by the other objects in the cadidate space
  - In Satisfaction Problems, on the other hand, the only objective is to discover an qualified object with as little search effort as possible.

Solving Problems by searching

- Nodes that have been generated but haven't yet been explored as sometimes called open. On the contrary, the nodes that have already been expanded are called closed.

A couple of examples: Travelling salesman

- The travelling salesman problem. Here the goal is to find the cheapest tour (i.e cheapest path that visits every node once and only once and return to the initial node) in a graph with N node and each edge assigned a nonnegative cost.
- The TSP belongs to a class of problems called NP-hard for which all known algorithms require exponential time in the worst case. However the use of bounding functions can help find the optimal tour in much less time.

A couple of examples: Travelling salesman

- Consider the two paths ABC and AED shown on the figure below. The cost of the overall solution is the cost of the partial tour + the cost of completing the tour.

- However since finding the optimal completion is almost as hard as finding the entire optimal tour, if we want the search to be efficient, we need to turn to some heuristic estimate of the completion cost

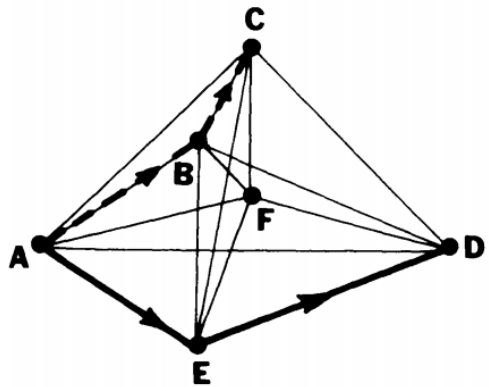


Figure 1.5

Two partial paths as candidates for solving the Traveling Salesman problem.

A couple of examples: Travelling salesman

- Given the heuristic estimates, the choice of which partial tour to extend first depends on which one of them, after we combine the cost of the explored part with the estimate of its completion offers the lower alternative.

- If at each stage, we select as extension the partial tour with the lowest estimated complete tour cost and if the heuristic function is optimistic (i.e it consistently underestimate the actual cost to complete a tour) then the first partial tour that is selected for extension and found to already be a complete tour is also a cheapest tour.

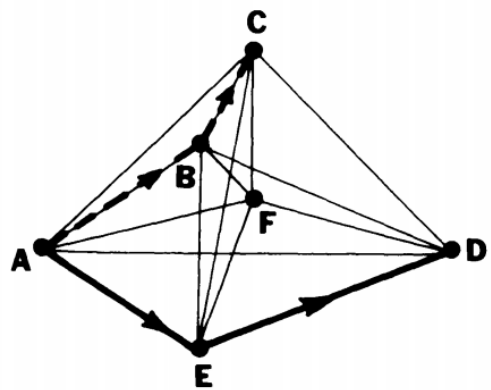


Figure 1.5

Two partial paths as candidates for solving the Traveling Salesman problem.

A couple of examples: Travelling salesman

- Two heuristics that have received the greatest attention are
  - The cheapest second degree graph going through all remaining nodes (including D and A below)
  - The minimum spanning tree (MST) through all remaining nodes (including D and A below)

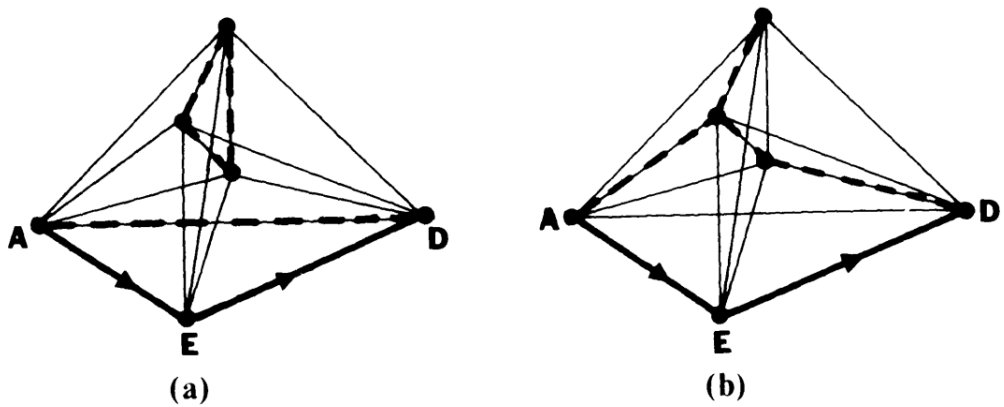


Figure 1.6

Two methods of optimistically estimating the completion cost of candidate $A \rightarrow E \rightarrow D$ in the Traveling Salesman problem of Figure 1.5. Part (a) The shortest degree-2 graph containing $A \rightarrow E \rightarrow D$. Part ($b$) The minimum spanning tree containing A, D, and all remaining (unlabeled) cities.

A couple of examples: Travelling salesman

- Cheapest second degree graph requires solving an optimal assignment problem (on the order of N^3 steps). The minimum spanning tree requires on the order of N^2 steps.

- Other (simpler) heuristics include taking as an estimate of the cost to complete the partial tour, the cost of the edge (or the two edges path) from the end of the tour to the initial node. Such functions however grossly underestimate the completion cost.

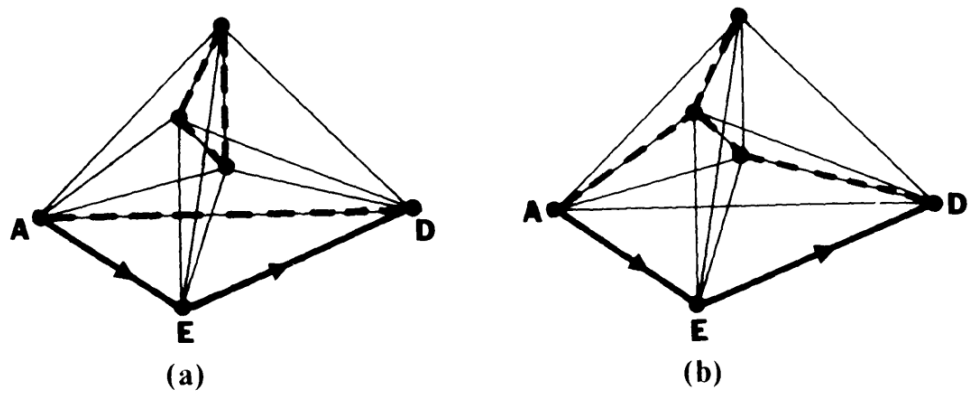


Figure 1.6

Two methods of optimistically estimating the completion cost of candidate $A \rightarrow E \rightarrow D$ in the Traveling Salesman problem of Figure 1.5. Part (a) The shortest degree-2 graph containing $A \rightarrow E \rightarrow D$. Part ($b$) The minimum spanning tree containing A, D, and all remaining (unlabeled) cities.

A couple of examples: Roadmap problem

- Given a map such as shown below, we want to find the shortest path between city A and city B assuming that each road on the map is specified by a label next to it.

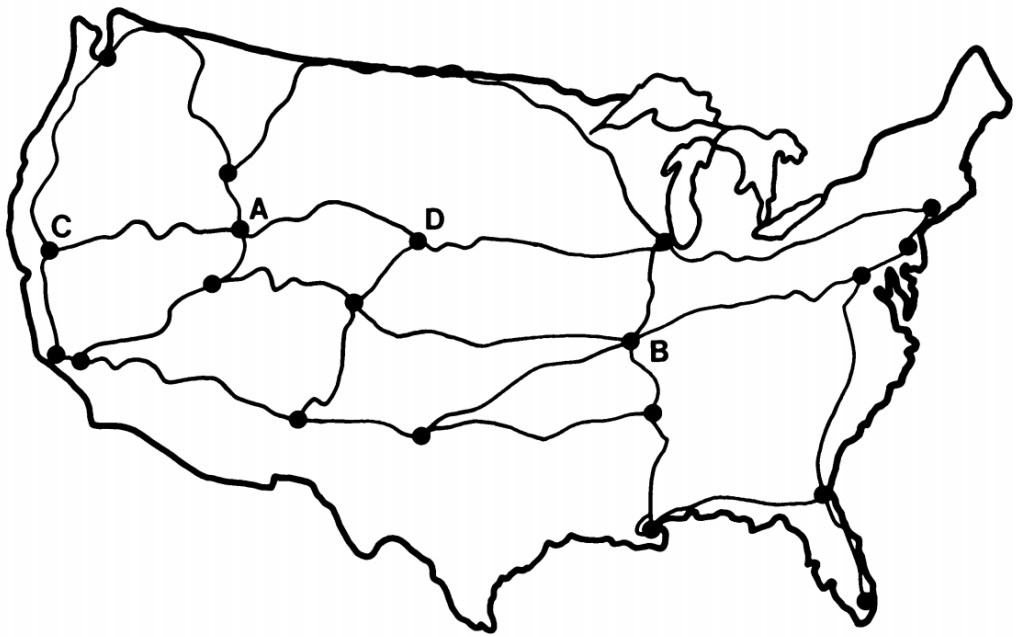


Figure 1.4
A road map, illustrating the use of heuristics in seeking the shortest path from *A* to *B*.

A couple of examples: Roadmap problem

- When given an actual map, we would rule out the roads that lead away from the general direction of the destination.

- An human observer looking at the map exploits vision machinery to estimate the Euclidean distances on the map and since the distance from D to B is shorted than the distance from C to B, city D appears as a more promising candidate.

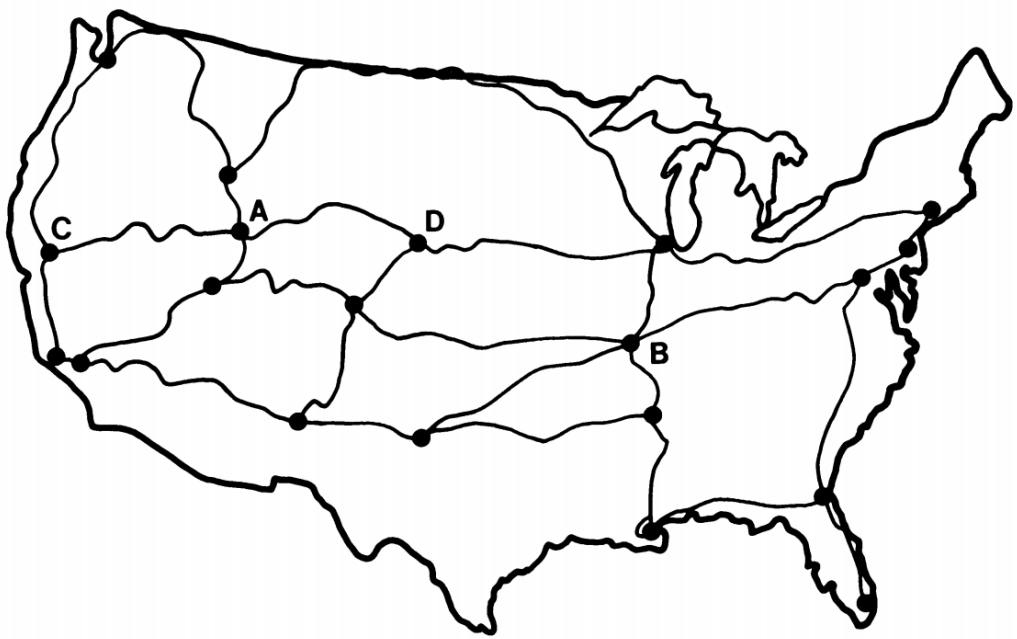


Figure 1.4
A road map, illustrating the use of heuristics in seeking the shortest path from *A* to *B*.

A couple of examples: Roadmap problem

- In the absence of a map (e.g. when given a table of pairwise distances between connected cities) we could attempt to simulate this extra information

- For example, as we can easily estimate air distances between cities from their coordinates, we can consider a heuristic function $h(i)$ which computes the air distance from city i to the goal city B.

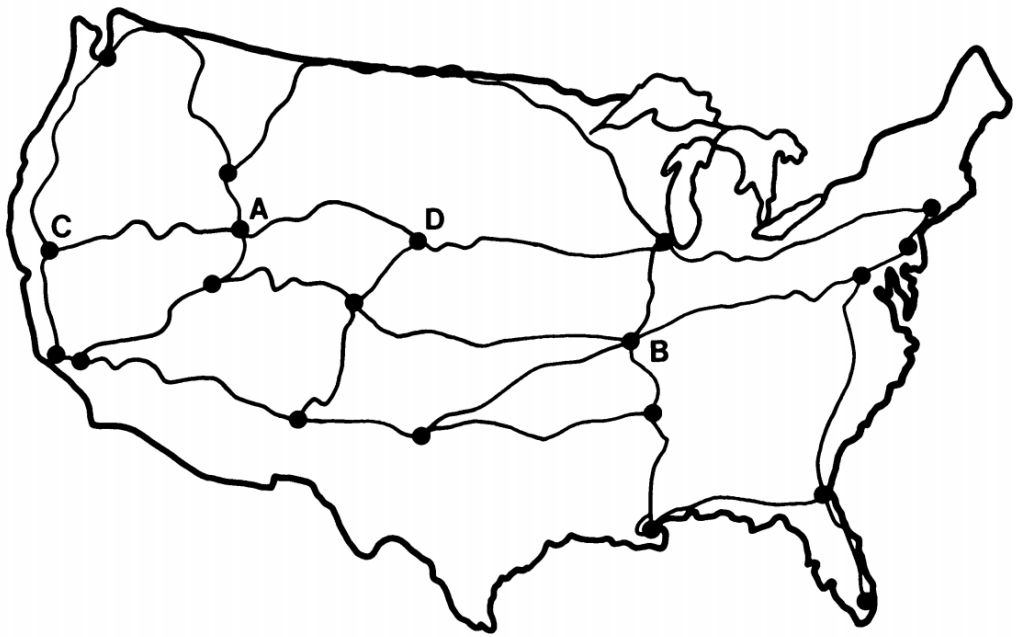


Figure 1.4
A road map, illustrating the use of heuristics in seeking the shortest path

A couple of examples: 8 queens problem

- The goal of the 8 queens problem is to place eight queens on a chess board such that no queens can attack another. This is equivalent to placing the queens so that no row, column or diagonal contains more than one queen

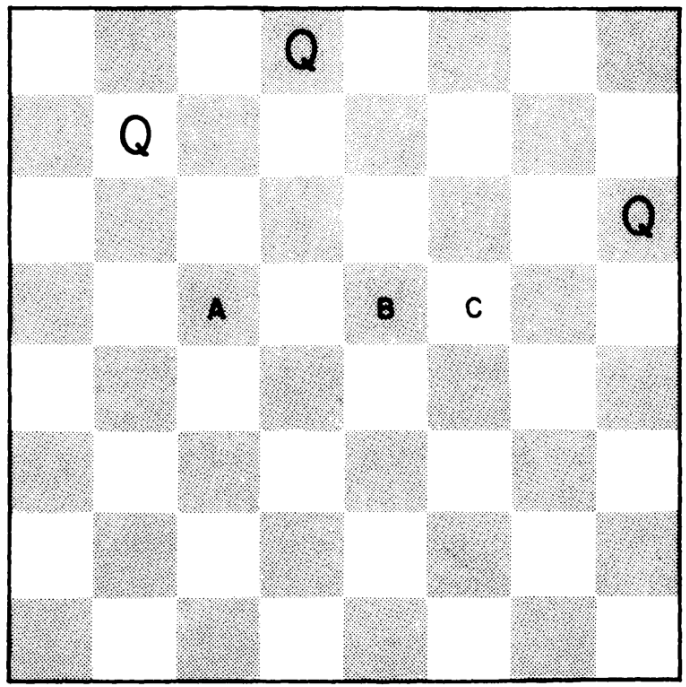


Figure 1.1
A typical decision step in constructing a solution to the 8-Queens problem.

A couple of examples: 8 queens problem

- As for the other problems, we can forgo the hope of obtaining the solution in one step and proceed step by step, in an incremental manner.

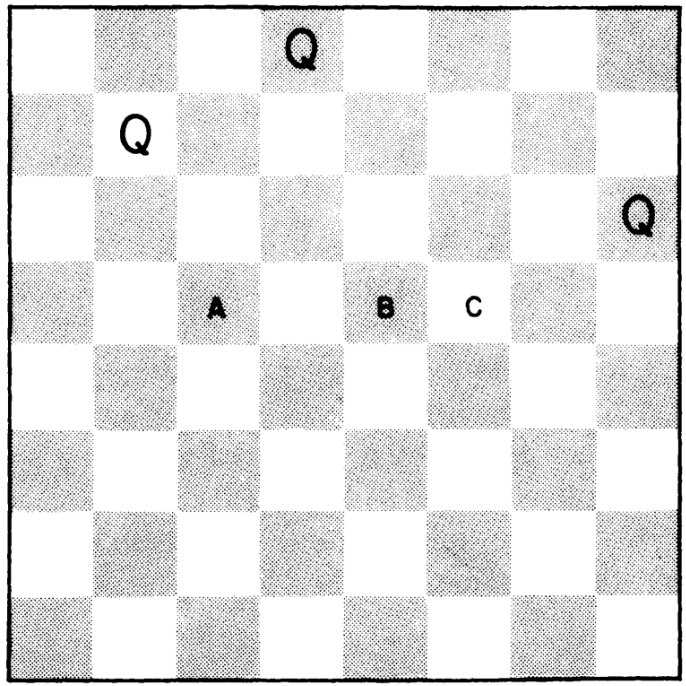


Figure 1.1
A typical decision step in constructing a solution to the 8-Queens problem.

A couple of examples: 8 queens problem

- One approach could be to start from an arbitrary arragement of the 8 queens that we would then transform iteratively, going from one board configuration to another, until the queens are adequately dispersed. The transformation should be systematic so that we do not apply the same transformation twice.

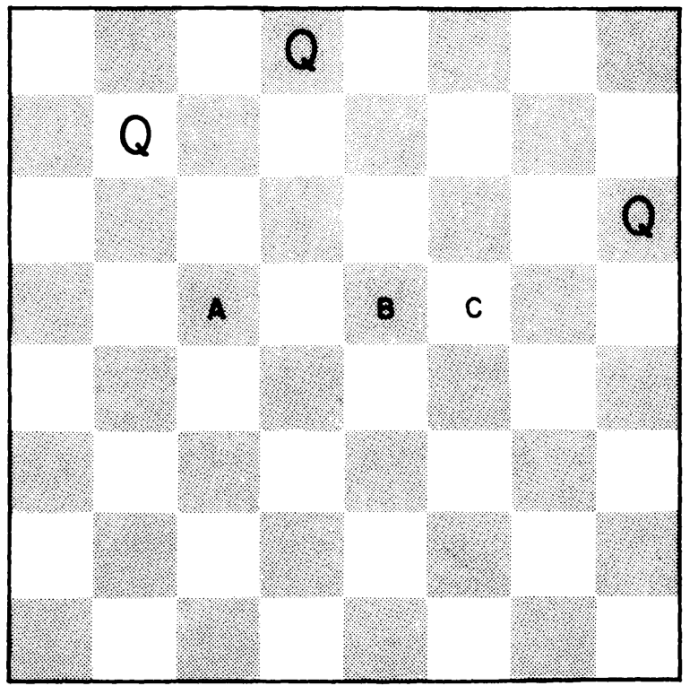


Figure 1.1
A typical decision step in constructing a solution to the 8-Queens problem.

A couple of examples: 8 queens problem

- An alternative would be to start with an empty board, then attempt to place the queens one at a time. This way we already rule out violations of the problem constraints. Since there can be only one queen in each row, we can assign the first queen to the first row, the second queen to the second row and so on

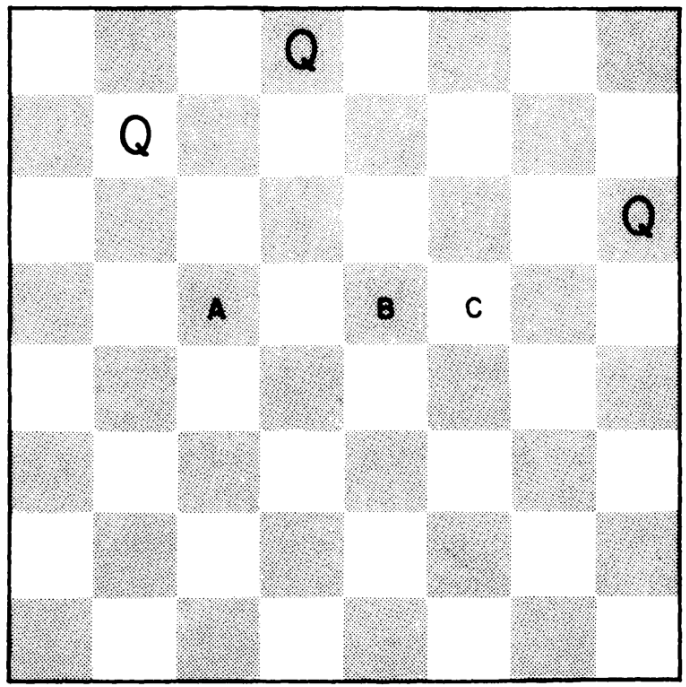


Figure 1.1
A typical decision step in constructing a solution to the 8-Queens problem.

A couple of examples: 8 queens problem

- Assume that we have positioned three queens as below and wonder whether we should position the fourth on in A, B or C. A heuristic in this case would have to determine, at least tentatively, which of the three positions appears to have the highest chance of leading to a satisfactory solution.

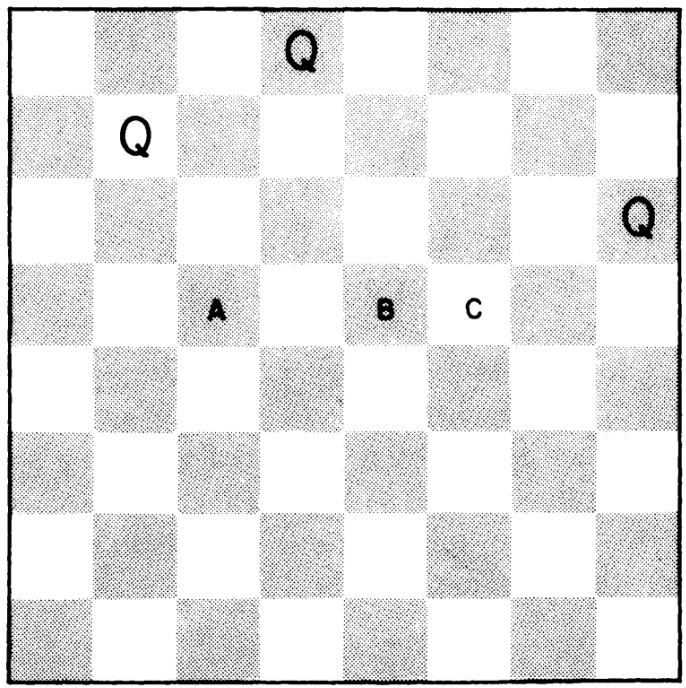


Figure 1.1
A typical decision step in constructing a solution to the 8-Queens problem.

A couple of examples: 8 queens problem

- We could define a first heuristic by preferring candidate solutions that leaves the highest number of unattacked cells on the board (i.e. to be able to place the remaining queens, we want to leave as many options as possible for future additions). If we let $f(\cdot)$ denote the number of unattacked cells, we get $f(A) = 8$, $f(B) = 9$ and $f(C) = 10$.

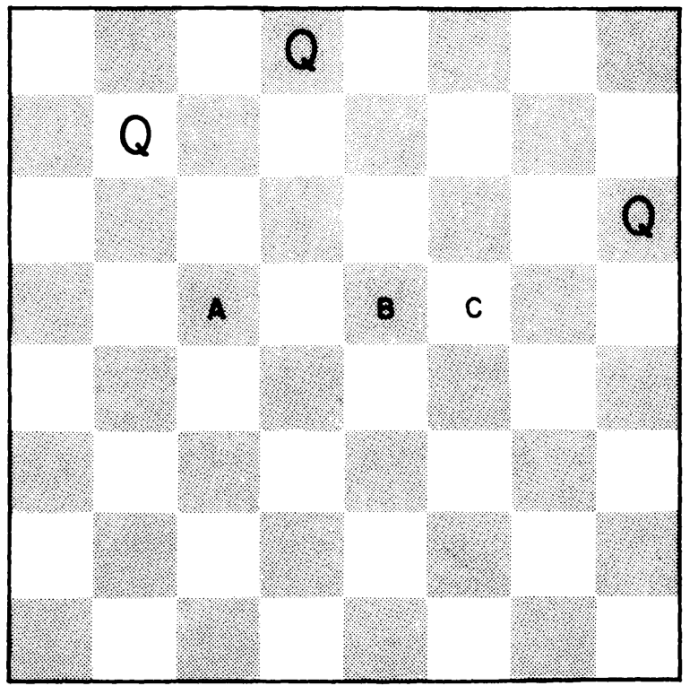


Figure 1.1
A typical decision step in constructing a solution to the 8-Queens problem.

A couple of examples: 8 queens problem

- A more sophisticated heuristic would focus on the rows with the smallest number of unattacked cells are those rows are more likely to provide unattacked cells in the future. We should then choose as our next position, the position that maximizes such rows. Using this as our heuristic, we get $f'(A) = f'(B) = 1$ and $f'(C) = 2$

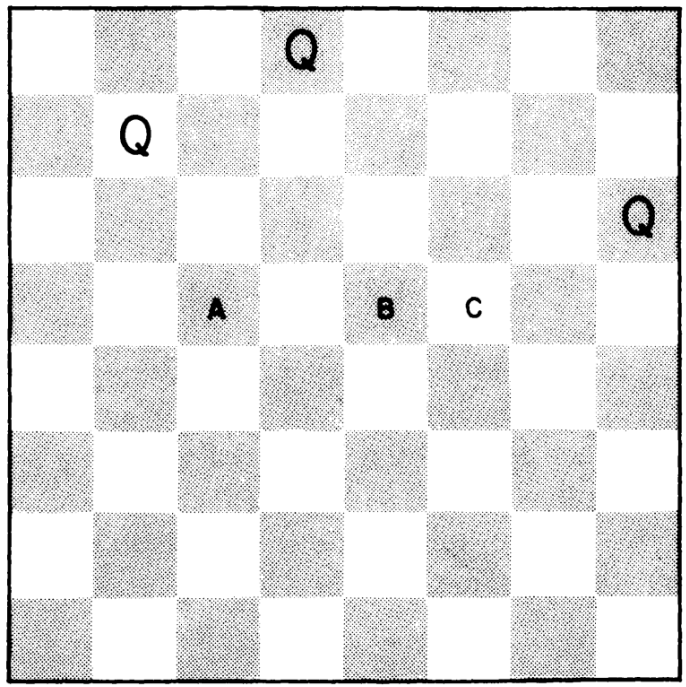


Figure 1.1
A typical decision step in constructing a solution to the 8-Queens problem.

A couple of examples: 8 queens problem

- A more sophisticated heuristic would focus on the rows with the smallest number of unattacked cells are those rows are more likely to provide unattacked cells in the future. We should then choose as our next position, the position that maximizes such rows. Using this as our heuristic, we get $f'(A) = f'(B) = 1$ and $f'(C) = 2$

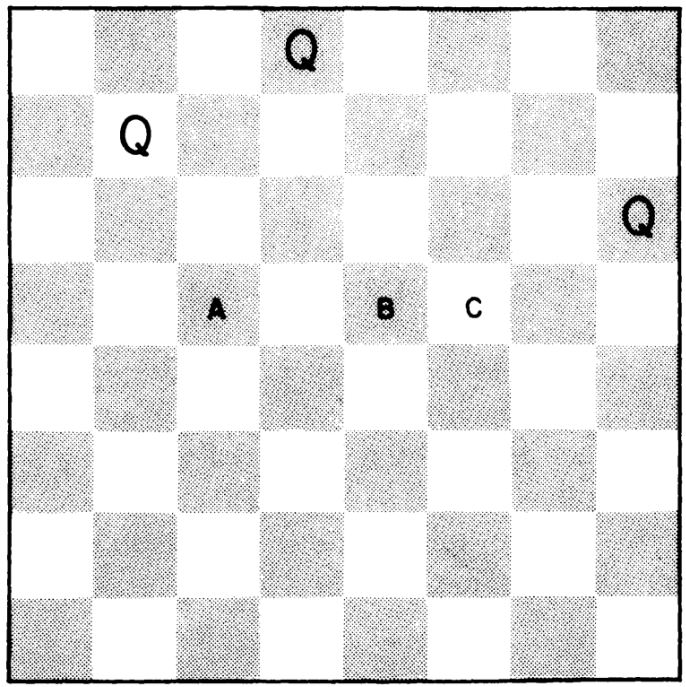


Figure 1.1
A typical decision step in constructing a solution to the 8-Queens problem.

A couple of examples: Counterfeit coin problem

- We are given 12 coins, one of which is know to be heavier or lighter than the rest. Using a two-pan scale, we must find the counterfeit coin and determine whether it is light or heavy in no more than three tests
- To solve the counterfeit coin problem, we must find a weighing strategy, i.e. a prescription of what to weigh first , what to weight second for each possible outcome of the first weight, and finally an indication of which coin is counterfeit.
- This coin can be solved by an exhaustive enumeration of all decisions and all possible test outcomes. The use of heuristics is to focus attention on the most promising weighing strategies and to attempt to find a solution without exploring all possible strategies

A first algorithm: Hill climbing

- In terms of graphs, hill climbing strategy amounts to repeatedly expanding a node, inspecting its newly generated successors, and choosing and expanding the best among those successors while retaining no further references to the father or siblings.
- When we come to a local maximum (a node more valuable than any of its successors), no further improvements is possible by local perturbations and the process must terminate without reaching a solution.
- In such procedure, the only way to free ourselves from a deadlock is to start afresh from what we hope is a completely new node, thus risking to violate an important principle in systematic search: do not turn any stone more than once.

A first algorithm: Hill climbing

- The strategy is called irrevocable, because the process does not permits us to shift attention back to the previously suspended alternatives.
- Hill climbing is a useful strategy when we possess a highly informative guiding function that keeps up away from local maxima, ridges and plateaux, and that can lead us quickly toward the global peak.

Uninformed Search Methods: DFS

- We now would like to bring our strategy in line with the requirements of systematization
- If for some reason a given search avenue is chosen for exploration, the other candidate alternatives should not be discarded but should be keptin reserve in case the avenue chosen fails to produce an adequate solution.
- In Depth First Search (DFS), priority is given to nodes at deeper levels of the the search graph. That is each node chosen for exploration gets all its successors generated before another node is explored
- Such a policy can be dangerous when implemented on large graphs, especially those of infinite depth.

Uninformed Search Methods: DFS

- To make sure that the search does not run indefinitely, Depth First Search Algorithms are usually equipped with a depth bound - a stopping rule that, when triggered, returns the algorithm's attention to the deepest alternative not exceeding this bound
- The program backtracks under one of two conditions:
  - The depth bound is exceeded
  - A node is recognized as a dead end

Uninformed Search Methods: DFS

1. Put the start node on OPEN
2. If OPEN is empty, exit with failure, otherwise continue
3. Remove the topmost node from OPEN and put it on CLOSED. Call this node n
4. If the depth of n is equal to the depth bound, clean up CLOSED and go to step 2. Otherwise continue
5. Expand n, generating all its successors. Put these successors (in no particular order) on top of OPEN and provide for each a pointer back to n
6. If any of these successors is a goal node, exit with the solution obtained by tracing back through its pointers. Oterhwsie continue.
7. If any of the successors is a dead end, remove it from OPEN and clean up CLOSED.
8. Go to step 2

Uninformed Search Methods: Backtracking

1. Backtracking is a version of DFS that applies the last in first out policy for node generation instead of node expansion.
2. When a node is selected first for exploration, only one of its successors is generated and the newly generated node, unless it is found to be a terminal or a dead end, is again submitted for exploration.
3. If the generated node meets some stopping criterion, the program backtracks to the closest unexpanded ancestor, that is, an ancestor still having ungenerated successors.

Uninformed Search Methods: backtracking

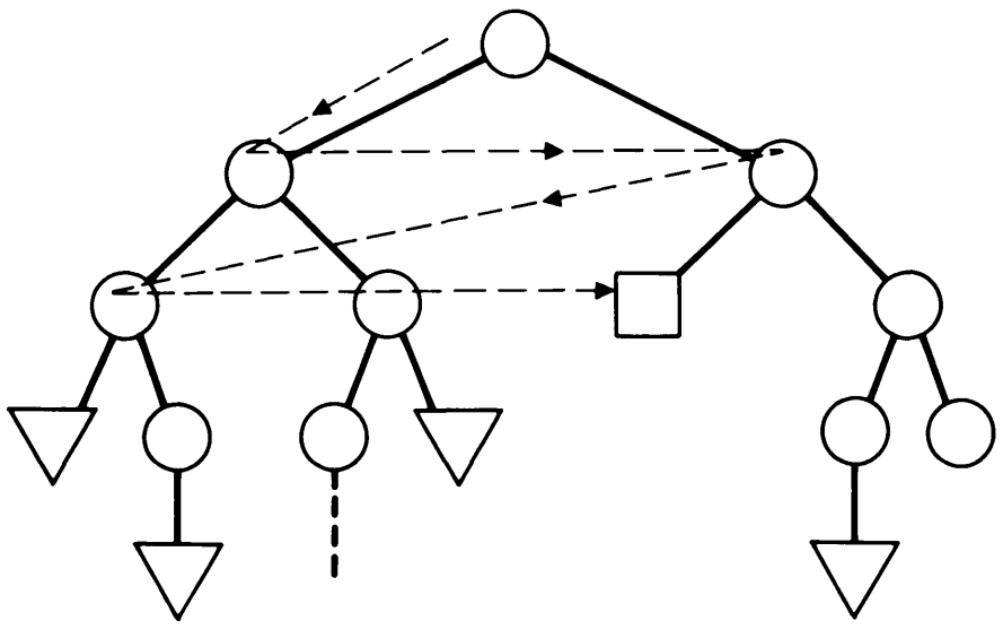
1. Put the start node on OPEN
2. If OPEN is empty, exit with failure, otherwise continue
3. Examine the topmost node from OPEN and call it n
4. If the depth of n is equal to the depth bound or if all the braches emanating from n have already been traversed, remove n from OPEN and go to step 2. Otherwise continue
5. Generate a new successor of n (along a branch not previously traversed), call it n'. Put n' on top of OPEN and provide a pointer back to n
6. Mark n to indicate that the branch (n, n') has been traversed
7. If n' is a goal node, exit with the solution obtained by tracing back through its pointers; otherwise continue
8. If n' is a dead end, remove it from OPEN
9. Go to step 2

Uninformed Search Methods: Backtracking

- The main advantage of backtracking over depth first search lies in achieving an even greater storage in memory
- Instead of retaining all the successors of an expanded node, we only store a single successor at any given time.

Uninformed Search Methods: Breadth First Strategies

- As opposed to depth first search, assign a higher priority to nodes athe shallowest levels of the searched graph
- Instead of Last In First Out (LIFO) policy, Breadth First Search is implemented by a First-In-First-Out (FIFO) policy, giving first priority to the nodes residing on OPEN for the longest time



(b)

Uninformed Search Methods: BFS

1. Put the start node on OPEN
2. If OPEN is empty, exit with failure, otherwise continue
3. Remove the topmost node from OPEN and put it on CLOSED. Call this node n
4. If the depth of n is equal to the depth bound, clean up CLOSED and go to step 2. Otherwise continue
5. Expand n, generating all its successors. Put these successors (in no particular order) at the bottom of OPEN and provide for each a pointer back to n
6. If any of these successors is a goal node, exit with the solution obtained by tracing back through its pointers. Oterhwsie continue.
7. If any of the successors is a dead end, remove it from OPEN and clean up CLOSED.
8. Go to step 2

Additional alternatives and improvements

- Other uninformed alternatives include Depth limited Search (impose a cutoff on the maximum depth of a path), Iterative Deepening Search (successively investigate all depth limits, starting with depth 0, then 1,...), Bidirectional search (simultaneously search both forward from the initial state and backward from the goal, stop when the two searches meet in the middle)

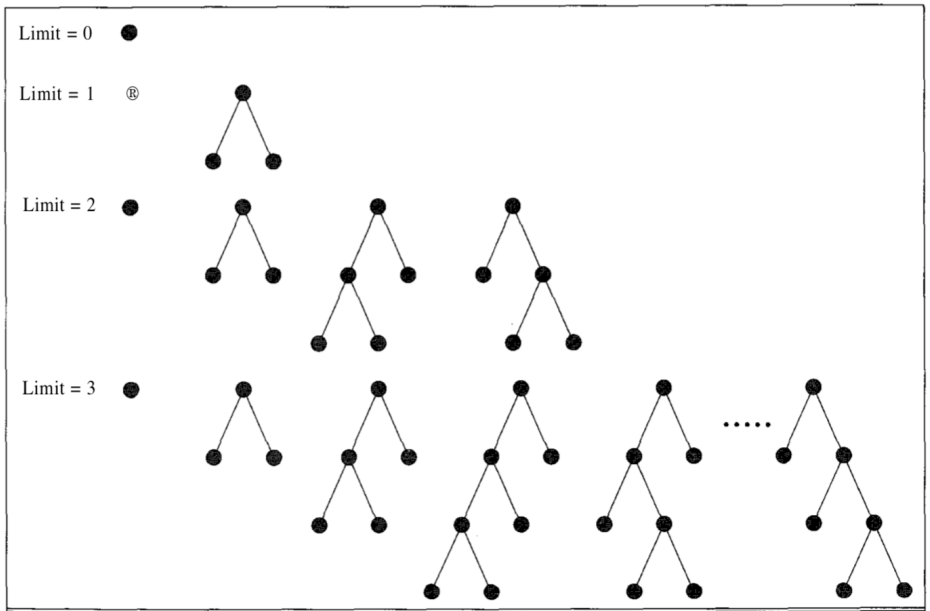


Figure 3.16 Four iterations of iterative deepening search on a binary tree.

Uninformed Search Methods: Breadth First Strategies

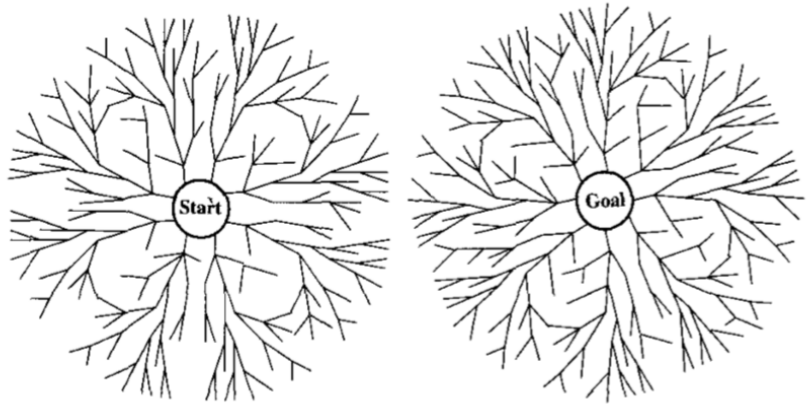


Figure 3.17 A schematic view of a bidirectional breadth-first search that is about to succeed, when a branch from the start node meets a branch from the goal node.

Informed Search Methods: Best First

- The promise of a node can be encoded in various ways. One way is the difficulty of solving the problem underlying that node, another way is to estimate the quality of the set of candidate solutions encoded by the node. Finally we could consider the amount of information we anticipate to gain by expanding the node.
- In all these approaches, the promise of a node is estimated numerically by means of a heuristic evaluation function $f(n)$

Informed Search Methods: Best First (I)

1. Put the start node s on a list called OPEN of unexpanded nodes
2. If OPEN is empty, exit with failure; no solution exists
3. Remove from OPEN a node n at which f is minimum (break ties arbitrarily) and place it on a list called CLOSED to be used for expanded nodes
4. Expand node n, generating all its successors with pointers back to n
5. If any of n's successors is a goal node, exit successfully with the solution obtained by tracing the path along the pointers from the goal back to s

Informed Search Methods: Best First (II)

6. For every successor n' of n:
   - 6.1 Calculate $f(n')$
   - 6.2 If n' was neither on OPEN nor on CLOSED,a dd it to OPEN. Attach a pointer from n' back to n. Assign the newly computed $f(n')$ to node n'
   - 6.3 If n' already resided on OPEN or CLOSED, compare the newly computed $f(n')$ with the value previously assigned to n'. If the old value is lower, discard the newly generated node. If the new value is lower, substitute it for the old (n' now points back to n instead of to its previous predecessor). If the matching node n' resided on CLOSED, move it back to OPEN
7. Go Back to Step 2

Informed Search Methods: A*

- We now use $g(n)$ to encode the cost of the current path from s (source) to n with $g(s) = 0$
- We let $h(n)$ to denote an estimate fo the $h^*(n)$, such that $h(\gamma) = 0$
- We also use f^* to denote the sum $f^*(n) = g^*(n) = h^*(n)$ (optimal cost over all solution paths constrained to go through n)

Informed Search Methods: A*

6. Put the node s on OPEN
7. If OPEN is empty, exit with failure
8. Remove from OPEN and place on CLOSED a node n for which f is minimum
9. If n is a goal node, exit successfully with the solution obtained by tracing back the pointers from n to s
10. Otherwise expand n, generating all its successors, and attach to them pointers back to n. For every successor n' of n
    - 10.1 If n' is not already on OPEN or CLOSED, estimate $h(n')$ (estimate of the cost of the beth path from n' to some goal node) and calculate $f(n') = g(n') + h(n')$ where $g(n') = g(n) + c(n, n')$ and $g(s) = 0$
    - 10.2 if n' is already on OPEN or CLOSED, direct its pointers along the path yielding the lowest $g(n')$
    - 10.3 If n' required pointer adjustment and was found on CLOSED, reopen it
11. Go to step 2.

Properties of heuristic methods

Definition (Completeness)

An algorithm is said to be complete if it terminates with a solution when one exists

Definition (Admissibility)

An algorithm is admissible if it is guaranteed to return an optimal solution whenever such a solution exists

Definition (Domination)

An algorithm A_1 is said to dominate another algorithm A_2 if every node expanded by A_1 is also expanded by A_2. Similarly, A_1 strictly dominates A_2 if A_1 dominates A_2 and A_2 does not dominate A_1. The expression "*more effcient than*" is sometimes used instead of "*dominates*".

Definition (Optimality)

An algorithm is said to be optimal over a class of algorithms if it dominates all members of this class.

Properties of heuristic methods

Theorem

A^* is complete on finite graphs

Proof.

The only possiblity for the search to complete is when we reach a goal node or when `open` is empty, in which case it is unsuccessful. However, since we assumed a single start node, as long as there exists an optimal path $P_{s \to \gamma}$, there will always be a node in `open` (recall that each time we expand a node, we add all the children of this node in `open`) $\square$