Chapter 2 Classical algorithms in Search and Relaxation

Chapter 2 overviews topics on the typical problems, data structures, and algorithms for inference in hierarchical and flat representations.

Part 1: Search on hierarchical representations
Heuristic search algorithms on And-Or graphs
– e.g. Best First Search, A*, Generalized Best First Search.

Part 2: Search on flat descriptive representations
-- e.g. Relaxation algorithm on line drawing interpretations

Part 1, Heuristic search in AI – a brief introduction

First, we introduce a few typical search algorithms in artificial intelligence. These search algorithms were studied in the 1960s-80s, and they search in hierarchic graph structures in a "top-down" manner with or without heuristic information. These techniques are very revealing. We should pay attention to their formulation, data structures, and ordering schemes which are very important ingredients in designing more advanced algorithms in visual inference. Unfortunately these issues have been largely ignored in the vision literature.

We will introduce four types of search algorithms:

Uninformed search	(blind search)
Informed search	(heuristic search, including A*)
*Adversarial search / reasoning	(game playing)
Search in And-Or graphs	

An Example

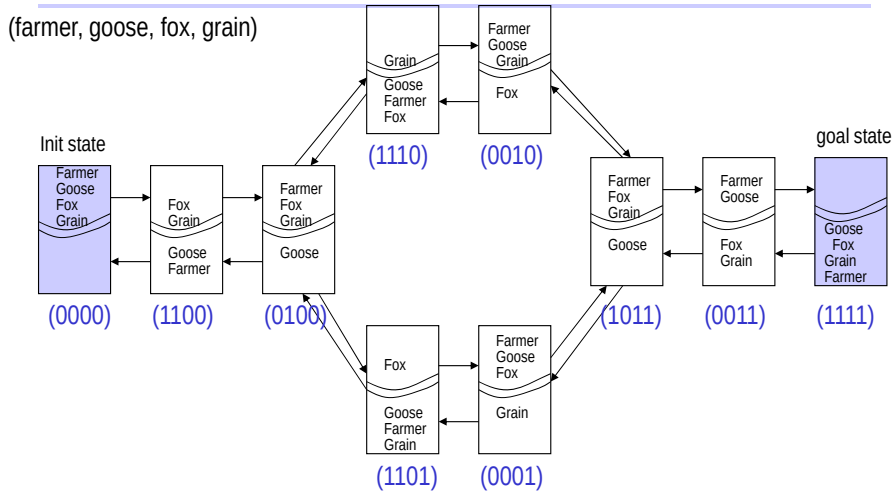
We start with a toy problem.

A *farmer* wants to move himself, a silver *fox*, a fat *goose*, and some Tasty *grain* across a river. Unfortunately, his *boat* is so tiny he can Take only one of his possessions across on any trip. Worse yet, an Unattended fox will eat a goose, and an unattended goose will eat Grain.

How can he cross the river without losing his possessions?

To solve such a problem in a computer, we must formulate it properly.

The State Graph in State Space



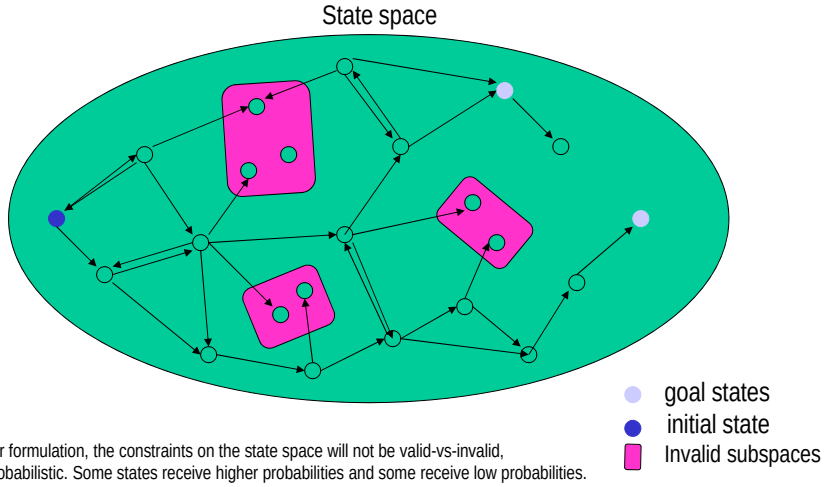
The "boat" is a metaphor. It has limited capacity and thus the states are connected locally.

Key elements in problem formulation

1. States (configurations, objects)
2. State space (there are constraints in the state space, e.g. fox eats goose)
3. Operators (available actions are often limited, e.g. the boat has *limited capacity*)
4. Initial state
5. Goal state(s) (Goal test)
6. Metrics (for multiple solutions, metrics measure their performance)

To design more effective algorithms, we need to design "big boats", which corresponds to large moves in the state space.

Example of state space and its transition graph



In later formulation, the constraints on the state space will not be valid-vs-invalid, but probabilistic. Some states receive higher probabilities and some receive low probabilities.

Criteria for Designing Search Algorithms

There are four criteria in designing a search algorithm

1. Completeness: is the algorithm guaranteed to find a solution if a solution exists?
2. Time complexity: this is often measured by the number of nodes visited by the algorithm before it reaches a goal node.
3. Space complexity: this is often measured by the maximum size of memory that the algorithm once used during the search.
4. Optimality: is the algorithm guaranteed to find an optimal solution if there are many solutions? A solution is *optimal* in the sense of minimum cost.

We say an algorithm is asymptotically optimal in space (or time) if the amount of space (time) required by the algorithm is within some additive or multiplicative factor of the minimum amount of space (time) required for the task in a computer.

Search in a tree (Or-tree)

Although the search is often performed in a graph in the state space, our study will focus on tree structured graph for two reasons:

1. It is convenient and revealing to analyze algorithm performance on trees. We can do the analysis on trees, and then generalize it to graphs.
2. If a search algorithm does not visit nodes that were visited in previous steps, then its paths form a tree.

Properties of a tree:

1. A tree is a connected graph with no loop: for a tree of n nodes, it always has $n-1$ edges.
2. Root, parent, children, terminal node/leaf, non-terminal/internal node.
3. Out number, branching factor b
4. Depth d
5. For a complete tree with depth d and branching factor b, there are b^d leaves and $1+ b^2 + \ldots \ldots + b^{d-1} = (b^d - 1)/(b-1)$ non-terminal nodes. Thus for $b>1$, the leaves outnumbers all the other nodes in the tree.

Category 1. Uninformed Search

The minimum information for searching a graph is the *goal test* --- a function that returns *true* when a node is a goal. If there is no other information available, the search algorithm is called uninformed search. This includes mainly

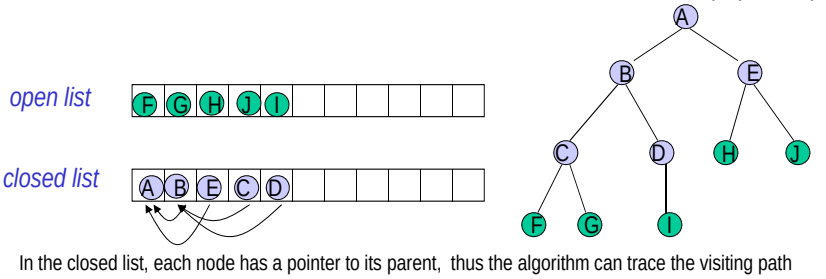
1. Depth first search (DFS)
2. Breadth first search (BFS)
3. Iterative deepening search

There are other variants, such as, limited depth search, bi-directional search, which are not required in this course.

Data Structure

In general, the algorithms operate on two lists:

1. An *open list* --- it stores all the leaf nodes of the current search tree. These nodes are to be visited (or expanded) in a certain order.
2. A *closed list* --- it stored all the internal nodes that have been visited (expanded).



In the closed list, each node has a pointer to its parent, thus the algorithm can trace the visiting path from a node to the root, at the final stage of extracting the solution.

Pseudo Code

General search algorithm:

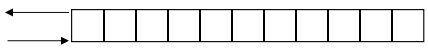
1. Initialize the *open list* by the initial node s_0, and set the *closed list* to *empty*.
2. Repeat
3. If the *open list* is *empty*, *exit* with failure.
4. Take the first node *s* from the *open list*.
5. If *goal-test(s) = true*, *exit* with success. Extract the path from *s* to s_0
6. Insert *s* in the *closed list*, *s* is said to be *visited* / *expanded*
7. Insert the *children of s* in the *open list* in a certain order.

Remark I: To avoid repeated visit, if a child of s appears in the closed/open lists, it will not be added.
Remark II: A node is said to be visited after it enters the closed list, not the open list.
Remarks III: In uninformed search, the space complexity is the *maximum length* that the open list once reached. The time complexity is measured by the *length* of the *closed list*.

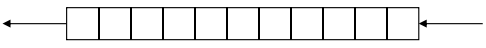
DFS and BFS

DFS and BFS differ only in the way they order nodes in the open list:

DFS uses a *stack*: it always puts the latest node on the top of the stack.



BFS uses a *queue*: it always puts the latest node at the end of the queue.



Complexities of DFS and BFS

For ease of analysis, we assume the graph is a complete tree with finite depth d and branching factor b. Suppose the goal node is at depth d.

For DFS,
The *best* case time complexity is $O(d)$: the goal is at the leftmost branch.
The *worst* case time complexity is $O(b^d)$: the goal is at the rightmost branch.
The space complexity is $O(db)$.

For BFS
The best/worst cases time complexity is $O(b^d)$.
The space complexity is $O(b^d)$.

But the DFS has a problem if the tree has depth $D >> d$. Even worse, if D can be infinite, Then it may never find the goal even if d is small --- not complete.

Iterative Deepening Search (IDS)

The DFS has advantage of less space/time complexity, but has a problem when $D>>d$. As d is often unknown to us, we can adaptively search the tree with incremental depth. This leads to IDS that combines the advantage of DFS and BFS.

1. Initialize $D_{max}=1$. The goal node depth d is unknown.
2. Repeat
3. Do a *DFS* starting from the root for a fixed depth D_{max},
4. Find a goal node, I.e. $d <= D_{max}$ then exit.
5. $D_{max} = D_{max} + \Delta$.

Iterative Deepening Search (IDS)

Time complexity: $O(b^d)$

$$\frac{b^2-1}{b-1}+\frac{b^3-1}{b-1}+\cdots+\frac{b^{d+1}-1}{b-1}=\frac{b^{d+2}-2b-bd+d+1}{(b-1)^2}$$

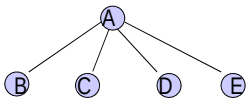
Space complexity: $O(db)$

So *IDS is asymptotically optimal in both space and time*.

Think why this makes sense. Apparently it wastes a lot of time to repeatedly visit nodes at the shallow levels of the tree. But this does not matter because the property 5 of tree.

Category 2. Informed Search

In some applications, an agent may have information about its goals or tasks. By using such information, we expect it can improve performance in space and time complexity.



Example:

Suppose at node A (see left figure), you are checking out in a supermarket. It has four lines B,C,D,E. You can decide which line to stand based on heuristic information of how long each line is.

Similar, when you are driving in a high way which has 4 lanes, which lane do you choose to drive at a given time period?

Heuristics

A poem quoted in Pearl 84.

> "Heuristics, Patient rules of thumb,
> So often scorned: Sloppy! Dumb!
> Yet, Slowly, common sense come" (ODE to AI)

In the 1980s, the probabilistic models are not well known to the CS search literature, Pearl viewed heuristics as ways to

"inform the search algorithm with simplified models"

In our understanding today, the heuristics are discriminative methods which inform or drive the Markov chain search in DDMCMC.

The Heuristics are "computational knowledge" in addition to the representational knowledge (models and dictionaries)

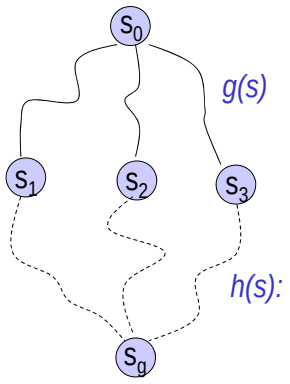
Informed Search

There are two new functions that an agent can explore:

g(s): this is a function that measures the "cost" it incurred from the initial node s_o to the current node s.

h(s): this is a function (or "budget") that estimates the forth-coming cost from s to a goal node s_g.

h(s) is called a heuristic function.



Pseudo Code for an Informed Search

"Best"-first-search (BFS) algorithm:

1. Initialize the *open list* by the initial node s_0,
   and set the *closed list* to *empty*.
2. Repeat
3. If the *open list* is *empty*, *exit* with failure.
4. Take the first node *s* from the *open list*.
5. Insert *s* in the *closed list*, *s* is said to be *visited* / *expanded*
6. If *goal-test(s) = true*, *exit* with success.
   Extract the path from *s* to s_0 in the closed list
7. Insert the *children of s* in the *open list* in
   an increasing order of a function x(s).

Choice of *x(s)*

The pseudo-code is called "best"-first search, because the algorithm always expands a node which it "think" is the "best" or most promising.

There are different choice for the function *x(s):*

1. *X(s)* can be *g(s).* In case, each edge has equal cost, then *g(s)=d(s)* is the depth. Then it reduces to a *breadth-first search*.

2. *X(s)* can be *h(s).* This is called the *greedy search.* It is similar to the depth first search and may not be complete

3. *X(s)* can be *f(s)=g(s) +h(s).* This is called a *heuristic search.*

Heuristic Search

By using the function f(s), a heuristic search algorithm can back-trace the most promising path

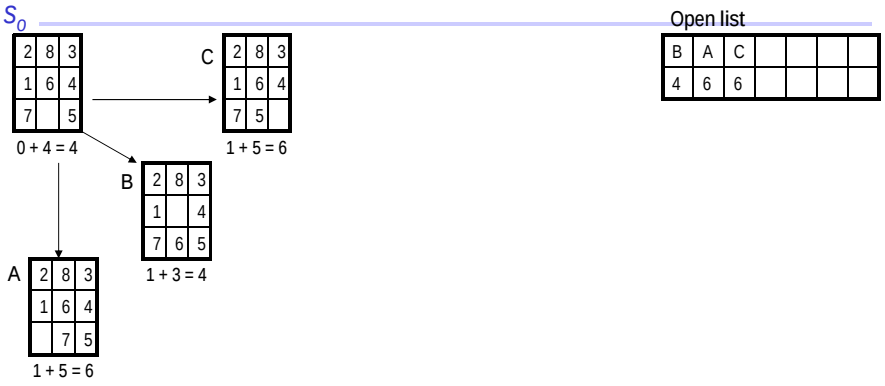
$$f(s) = g(s) + h(s).$$

How do we design h(s) so that the search can be effective

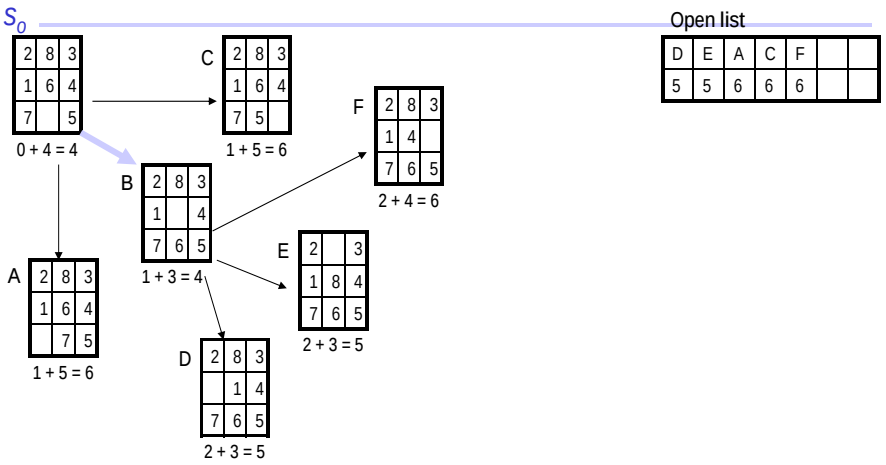
--- in terms of a small search tree ?

--- in terms of an optimal solution ?

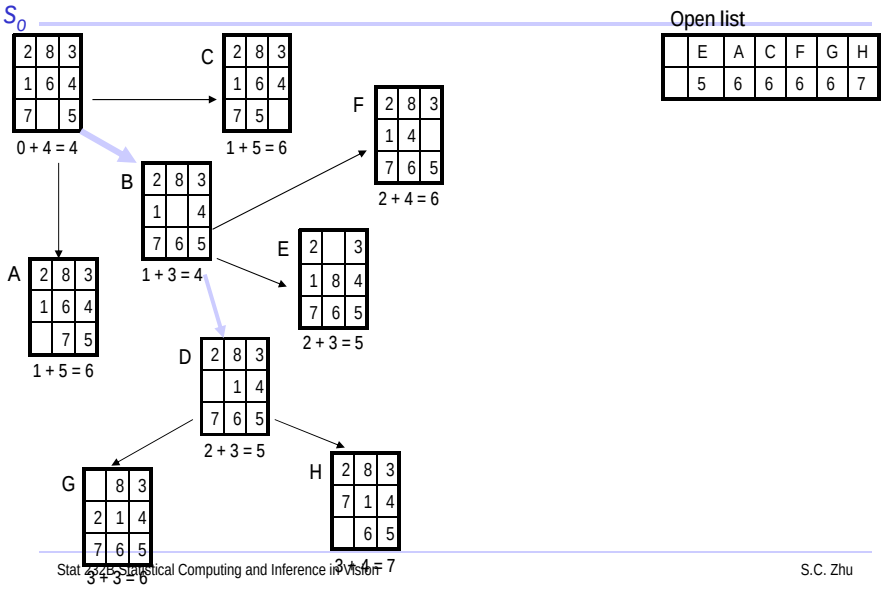
The 8-Puzzle Example



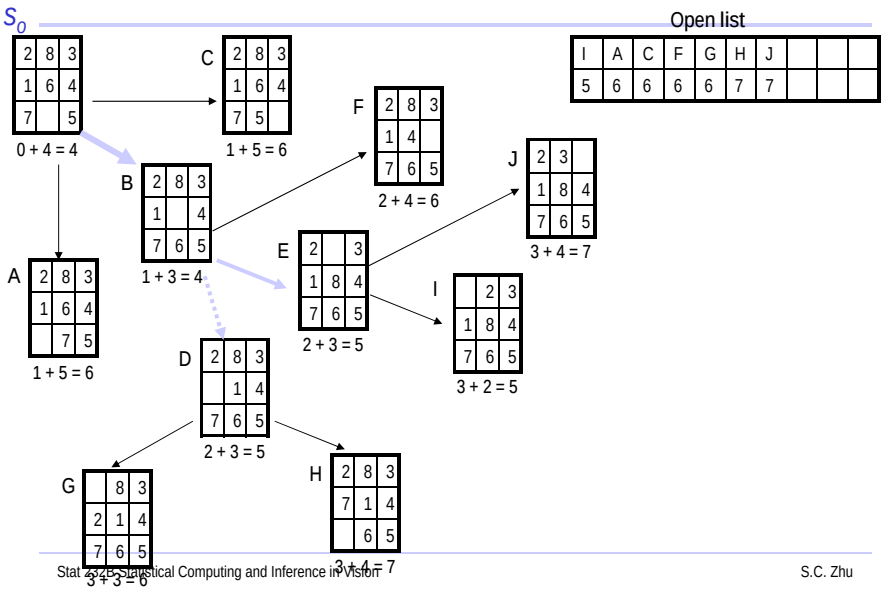
The 8-Puzzle Example



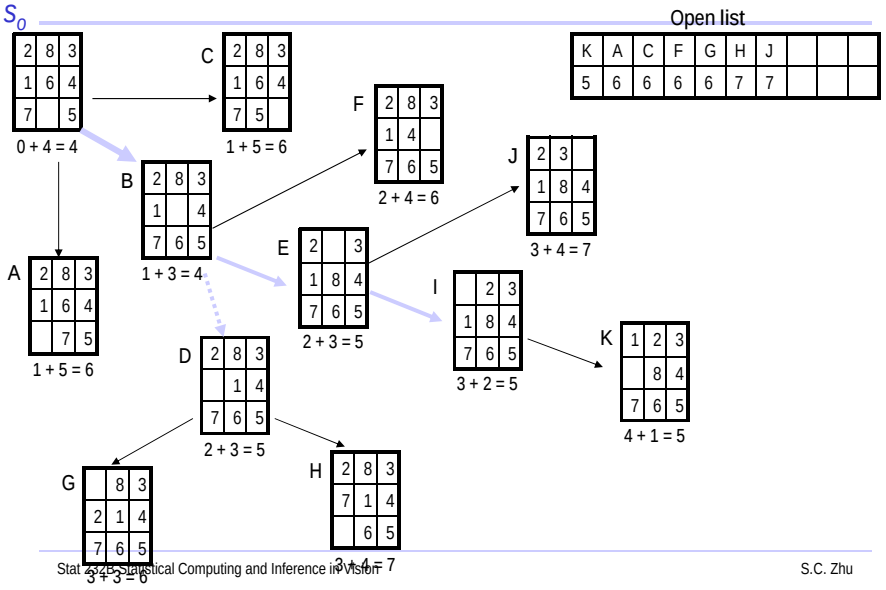
The 8-Puzzle Example



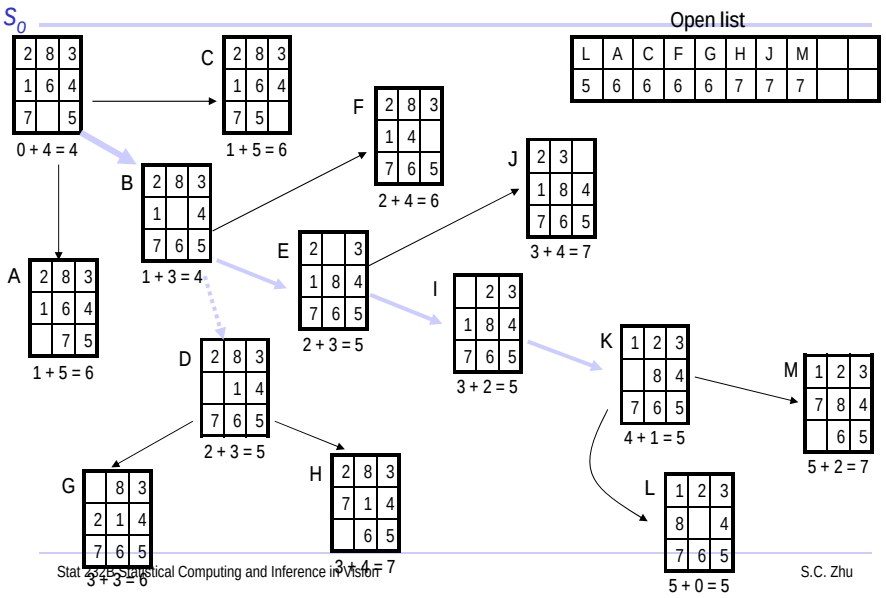
The 8-Puzzle Example



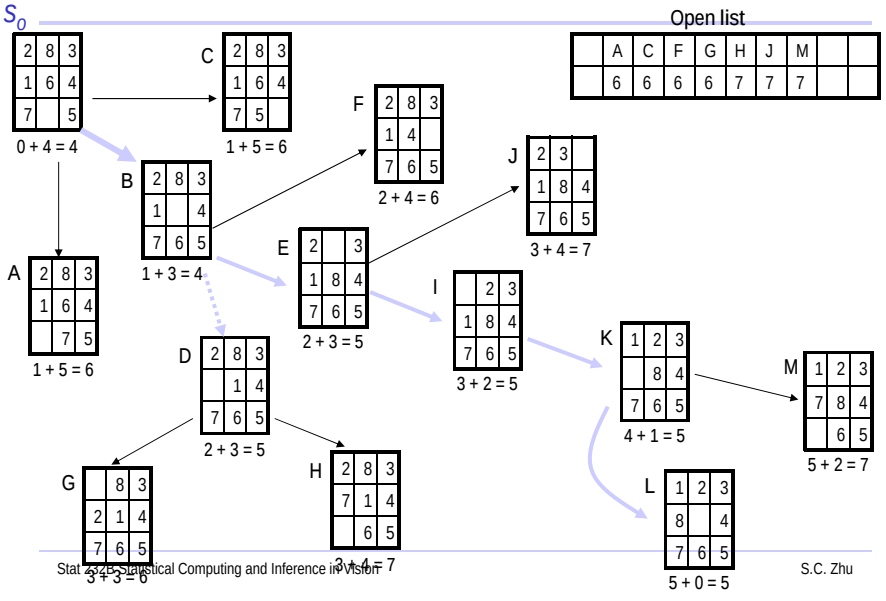
The 8-Puzzle Example



The 8-Puzzle Example



The 8-Puzzle Example



Admissible Heuristics

Can we design a heuristic function $h(s)$ so that the first found path is always the optimal (shortest or minimum cost) path?

Definition I: a heuristic function $h(s)$ is said to be admissible if it never over-estimates the true cost from s to a goal s_g --- $C(s, s_g)$, *I.e.*

$$h(s) <= C(s, s_g).$$

Definition II: a heuristic search algorithm is called an A*--algorithm if it uses an admissible heuristic function.

Definition III: we say $h_2(s)$ is more informed than $h_1(s)$ if

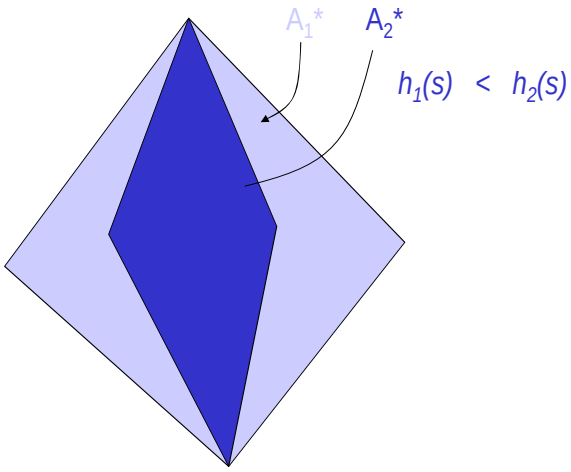
$$h_1(s) < h_2(s) <= C(s, s_g)$$

Some Good News

Proposition I:
An A* algorithm is always complete and optimal.

Proposition II:
Let A_1* and A_2* be two algorithms using heuristic functions $h_1(s)$ and $h_2(s)$ respectively, then the nodes searched by A_2* Is always a subset of those searched by A_1*, if $h_2(s)$ is more informed than $h_1(s)$, *I.e.* $h_1(s) < h_2(s) <= C(s, s_g)$

More Informed Searches Less Nodes



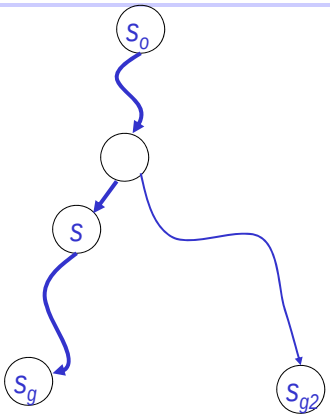
Proof of A* Optimality

Proof by refutation:

Let s_g be an optimal goal state with minimum cost $f^*(s_g)$. by refutation, suppose A* algorithm find a path from an initial state to a goal s_{g2} with cost $f(s_{g2}) > f^*(s_g)$. It could be that $s_{g2} = s_g$ and just the pathes found by A* has larger cost.

Then there must be a node s on the optimal path which is not chosen by A*. Since A* has expanded before s is expended. Thus it must be that $f(s) > f(s_{g2}) > f^*(s_g)$. Thus implies $h(s) > h^*(s)$. It is contradictory to the assumption of A* algorithm.

Proof of A* Optimality



Iterative Deepening A*

Obviously, the heuristic function reduces computational complexity. The extent to which the complexity decreases depends on how informed *h(s)* is. For example, if *h(s)=0*, the algorithm is clearly uninformed, and the A* algorithm degenerates to BFS.

We learn that BFS is actually an A* special case, and thus it is complete and optimal. But we know that BFS is notorious for space complexity, thus we doubt that A* may suffer from the same problem. Our worry is often true in real situation. To reduce the space complexity, we again introduce an iterative deepening algorithm called *Iterative Deepening A** (IDA*) this time. It employs a DFS algorithm with a bound for *f(s)*, and increases this bound incrementally.

Branch-and-Bound

Discussion the algorithm here and connection to Iterative deepening A*

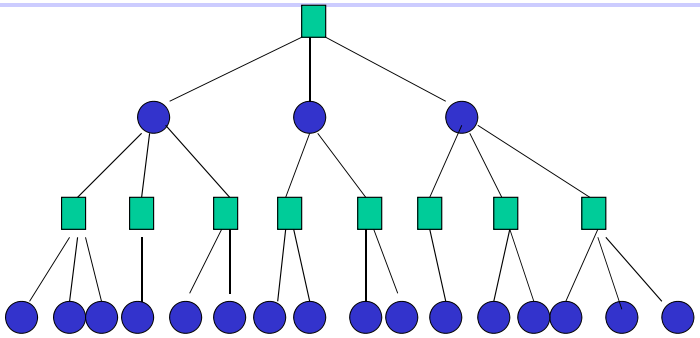
Category 3. **Adversarial Search

In this lecture, we introduce a new search scenario: game playing

1. two players,
2. zero-sum game, (win-lose, lose-win, draw)
3. perfect accessibility to game information (unlike bridge)
4. Deterministic rule (no dice used, unlike Back-gammon)

The problem is still formulated as a graph on a state space with each state being a possible configuration of the game.

The Search Tree



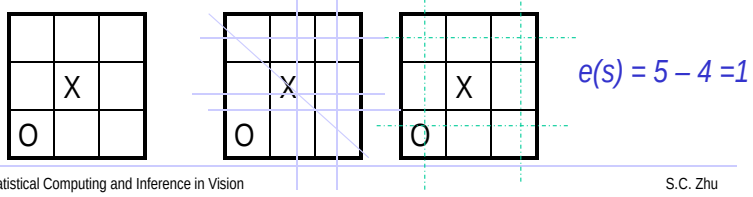
We call the two players Min and Max respectively. Due to space/time complexity, an algorithm often can afford to search to a certain depth of the tree.

Game Tree

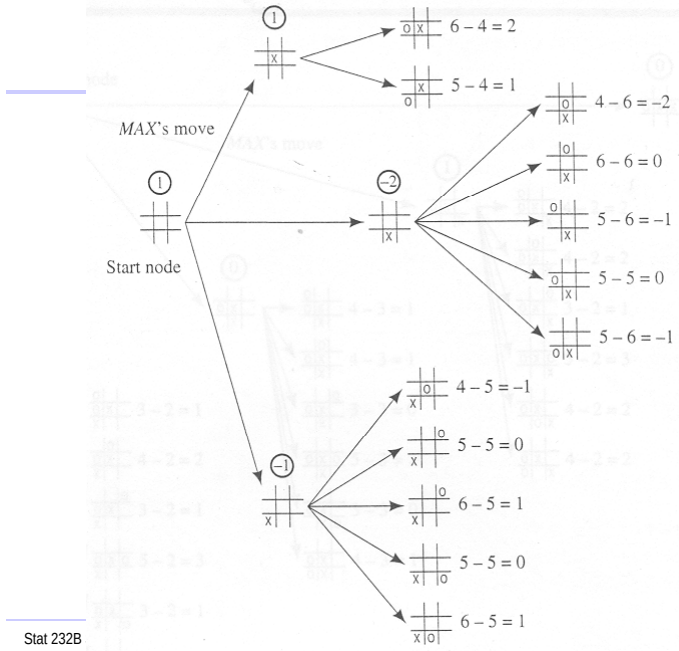
Then for a leave s of the tree, we compute a *static evaluation function* *e(s)* based on the configuration *s*, for example piece advantage, control of positions and lines etc. We call it static because it is based on a single node not by looking ahead of some steps. Then for non-terminal nodes, We compute the *backed-up values* propagated from the leaves.

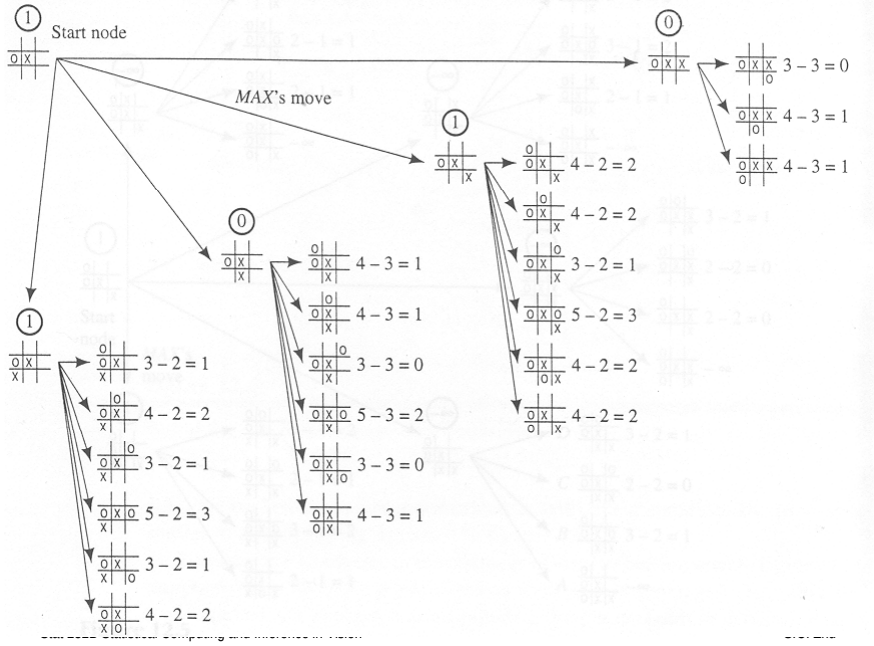
For example, for a tic-tac-toe game, we can select

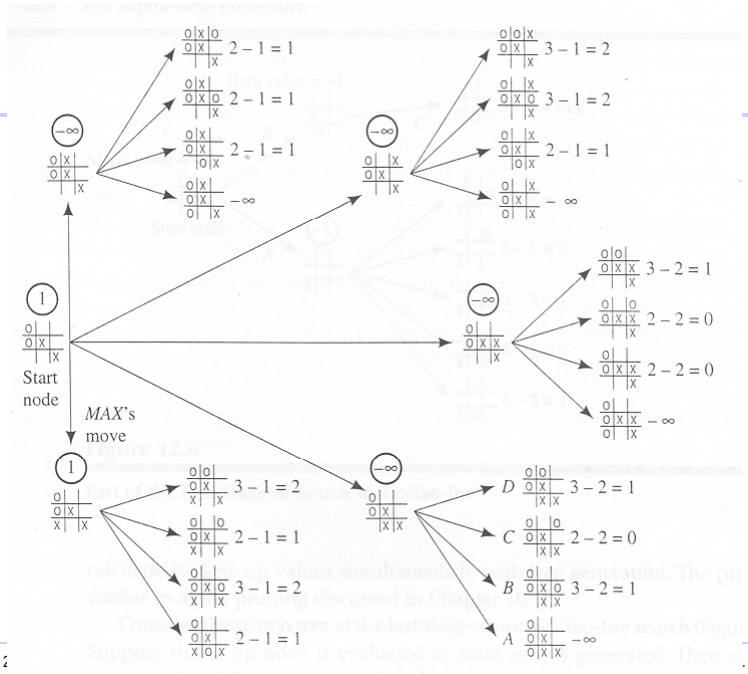
e(s) = # of open lines of MAX --- # of open lines of MIN

S = 

e(s) = 5 – 4 =1


MAX's move
Start node
6 – 4 = 2
5 – 4 = 1
4 – 6 = –2
6 – 6 = 0
5 – 6 = –1
5 – 5 = 0
5 – 6 = –1
4 – 5 = –1
5 – 5 = 0
6 – 5 = 1
5 – 5 = 0
6 – 5 = 1


Start node
MAX's move
3 – 3 = 0
4 – 3 = 1
4 – 3 = 1
4 – 2 = 2
4 – 2 = 2
3 – 2 = 1
5 – 2 = 3
4 – 2 = 2
4 – 2 = 2
4 – 3 = 1
4 – 3 = 1
3 – 3 = 0
5 – 3 = 2
3 – 3 = 0
4 – 3 = 1
3 – 2 = 1
4 – 2 = 2
3 – 2 = 1
5 – 2 = 3
3 – 2 = 1
4 – 2 = 2



Alpha-Beta Search

During the min-max search, we find that we may prune the search tree to some extent without changing the final search results.

Now for each non-terminal nodes, we define an α and a β values

A MIN nodes updates its β value as the minimum among all children who are evaluated. So the β value decreases monotonically as more children nodes are evaluated. Its α value is passed from its parent as a threshold.

A MAX nodes updates its α value as the maximum among all children who are evaluated. So the α value increases monotonically as more children nodes are evaluated. Its β value is passed from its parent as a threshold.

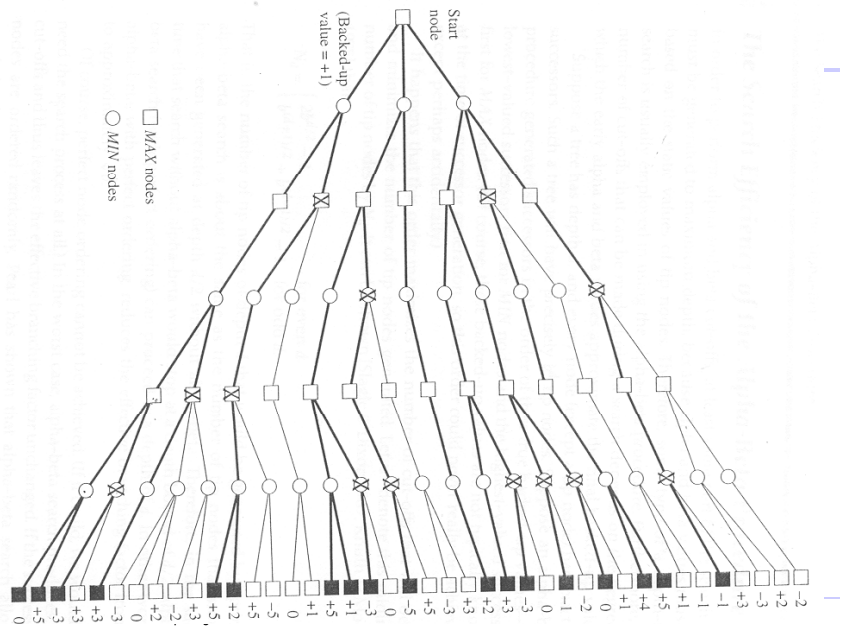
Alpha-Beta Search

We start the alpha-beta search with AB(root, α= - infinity , β=+infinity),
Then do a depth-first search which updates (α, β) along the way,
we terminate a node whenever the following cut-off condition is satisfied.

$$\beta <= \alpha$$

A recursive function for the alpha-beta search with depth bound D_{max}


```
Float:  AB(s, α, β)
{ if (depth(s) == Dmax)  return (e(s));      //  leaf node
  for  k =1 to b(s) do                       //  b(s) is the out-number of s
     {s' := k-th child node of s
        if (s is a MAX node)
          { α := max(α, AB(s', α, β))        // update α  for max node
            if ( β <= α)  return (α);        // β –cut
          }
        else
          {β := min(β, AB(s', α, β))         // update β  for min node
            if (β <= α)  return (β);         // α –cut
          }
      }     // end for
    if (s is a MAX node)  return (α);        // all children expanded
   else   return (β);
}
```

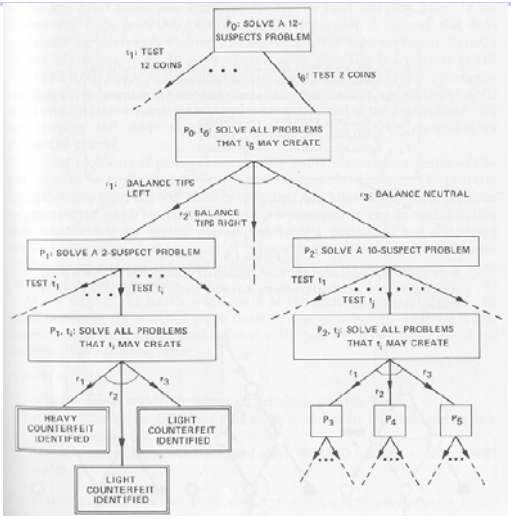


Category 4. Search in And-Or graphs

The 12 Counterfeit coin problem

Given 12 coins, one is known to be heavier or lighter than the others.
Find that coin with no more than 3 tests using a two-pan scale.

This generates the And-Or graph representation.



And-Or Graph is also called “hyper-graph”

The and-Or graph represents the decomposition of task into sub-tasks recursively.

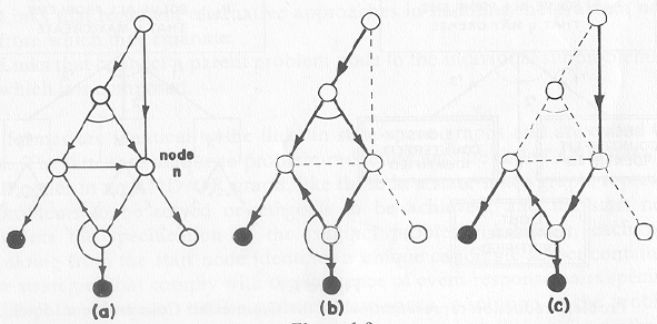


Figure 1.9
An AND/OR graph (a) and two of its solution graphs (b) and (c). Terminal nodes are marked as black dots.

Search in an And-Or Graph

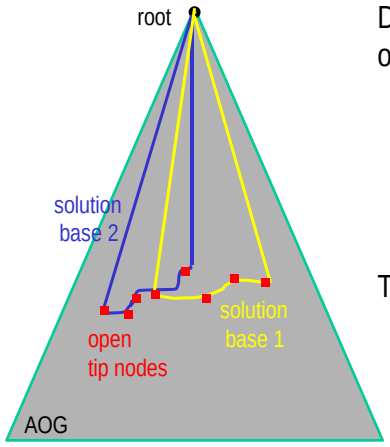
Important concepts:

1, *And-Or graph*: includes all the possible solutions. In an AoG, each Or-node represents alternative ways for solving a subtask, and each and-node represents a decomposition of a task into a number of sub-tasks. An Or-node is solvable if at least one of it children node is solvable, and an And-node is solvable only when all of its children nodes are solvable. An AoG may not be fully explicated at the beginning and is often expanded along the search process.

2, *Solution graph*: is a sub-graph of the AoG. It starts from the root node and makes choice at or-nodes and all of its leave nodes are solvable.

3, *Solution base graph*: a partial solution graph containing all the open nodes to be explored.

Search in an And-Or Graph



During the search, we need to maintain the open lists at two levels:

1, a list of the solution bases,

2, lists of open nodes for each solution bases.

The score f(s) = g(s) + h(s) for each node s is updated. A node s may be reachable from the root from multiple paths. Therefore, its score will have to be updated, and *rollback the updates* to its ancestor nodes.

Back-tracking in an And-Or tree

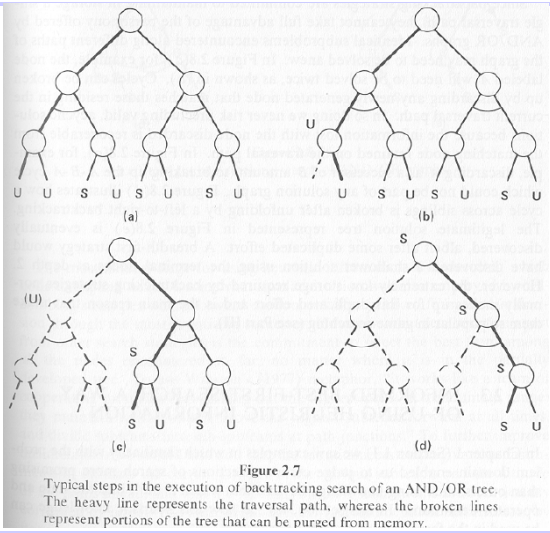


Figure 2.7
Typical steps in the execution of backtracking search of an AND/OR tree. The heavy line represents the traversal path, whereas the broken lines represent portions of the tree that can be purged from memory.

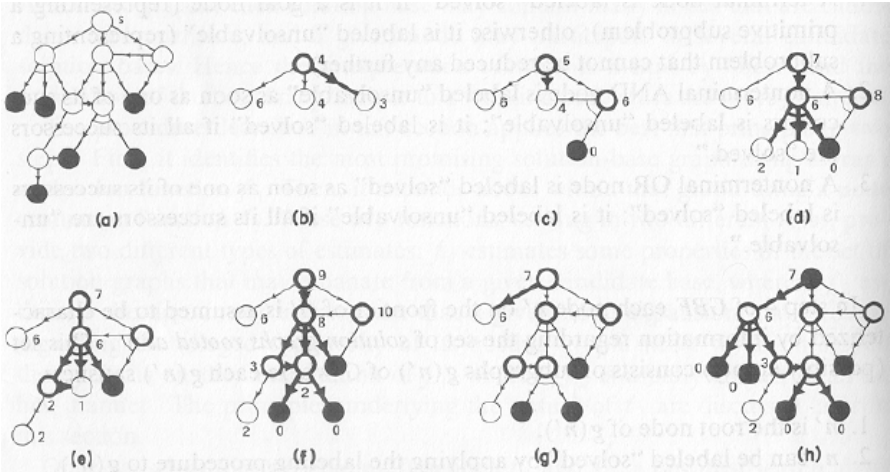
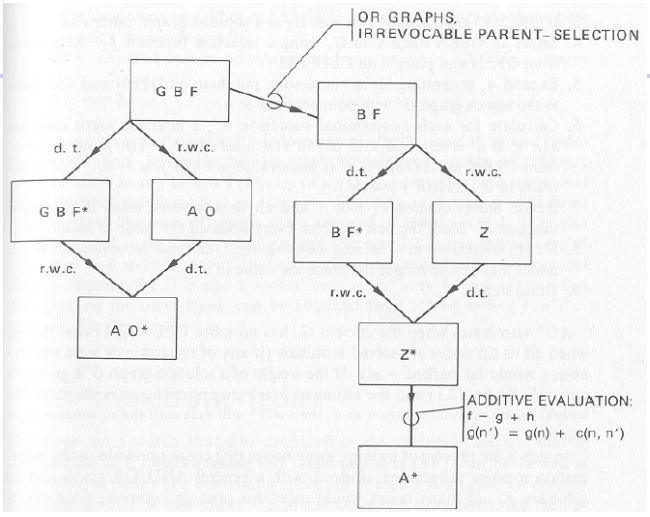


Figure 2.9

Successive steps in the execution of general-best-first (GBF) search on the implicit AND/OR graph of part (*a*). Solid circles represent solved nodes, heavy hollow circles nodes in CLOSED, and thin circles nodes in OPEN. The heavy lines stand, at each stage, for the current most promising solution base.

Relation map between various heuristic search algorithms



d.t.: delayed-termination, r.w.c.: recursive weight computation

Characterization of these problems

The And-Or graph search problem has the following components

1. An initial state
2. A number of "*operators*" to *generate* a set of new states (children / off-springs),
3. The Or-nodes for alternative ways
4. The And-nodes for sub-tasks that must be solved in combination
5. Metrics for each operator
6. Leaf nodes are solvable or unsolvable. In practice, each leaf node has a score for the likelihood.
7. The final solution is called a "solution graph".

Read more in the Pearl Chapter.