3 Search Spaces and Objective Functions. Complete Search Methods

3.1 Search Spaces and Objective Functions (1/4)

An instance I of a combinatorial search or optimisation problem Π determines a *search space* X of candidate solutions.

The computational difficulty in such problems arises from the fact that X is typically exponential in the size of I (= HUGE).

E.g. SAT:
Instance F = propositional formula on n variables $\{x_1, \ldots, x_n\}$.
Search space X = all truth assignments $t : \{x_1, \ldots, x_n\} \to \{0,1\}$.
Goal: find $t \in X$ that makes F true.

Size of X = 2^n points ($0/1$-vectors).

Search Spaces and Objective Functions (2/4)

Note that if SAT formulas are required to be in conjunctive normal form (as in e.g. 3-SAT), then it can also be viewed as an optimisation problem:

OPT-3-SAT:
Instance F = family of m 3-clauses on n variables $\{x_1, \ldots, x_n\}$.
Search space X = all truth assignments $t : \{x_1, \ldots, x_n\} \to \{0,1\}$.
Objective (cost) function: $c(t)$ = number of clauses not satisfied by t.
Goal: minimise $c(t)$.

Search Spaces and Objective Functions (3/4)

OPT-TSP:
Instance: An $n \times n$ matrix D of distances d_{ij} between n "cities".
Search space: X = all permutations ("tours") π of $\{1, \ldots, n\}$.
Cost function: $d(\pi) = \sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(1)}$.
Goal: minimise $d(\pi)$.

Note: Here $|X| = n!$. (More precisely: $|X| = (n-1)!/2$, if the starting points and orientations of tours are ignored.)

Search Spaces and Objective Functions (4/4)

OPT-SG (or SPIN GLASS GROUND STATE):
Instance: An $n \times n$ matrix C of "coupling constants" c_{ij} between n "spins" and an n-vector h ("external field").
Search space: X = all "spin configurations" $\sigma \in \{-1,1\}^n$.
Cost function ("Hamiltonian"):

$$H(\sigma) = -\sum_{\langle i,j \rangle} c_{ij}\sigma_i\sigma_j - \sum_i h_i\sigma_i.$$

Goal: minimise $H(\sigma)$.

Here again $|X| = 2^n$.

3.2 Complete Search Methods: Backtrack Search (1/2)

Bactrack search is a systematic method to search for a satisfying, or an optimal solution x in a search space X.

function backtrack(I:instance; x:partialsol):
    if x is a complete solution **then**
        return x
    else
        for all extensions $e_1, \dots e_k$ to x **do**
            $x' \leftarrow$ backtrack($I, x \oplus e_i$);
            if x' is a complete solution **then return** x'
        end for;
        return fail
    end if.

Backtrack Search (2/2)

For instance, in the case of SAT, each partial truth assignment $t : \{x_1, \dots, x_i\} \to \{0, 1\}$ has two possible extension e_0 and e_1: one assigns value 0 to variable x_{i+1} and the other assigns value 1.

In the case of TSP, the partial solutions could be nonrepeating sequences of cities (initial segments of tours), and the extensions could be choices of next city. (Also other arrangements are possible).

3.3 Backtrack Search in Games

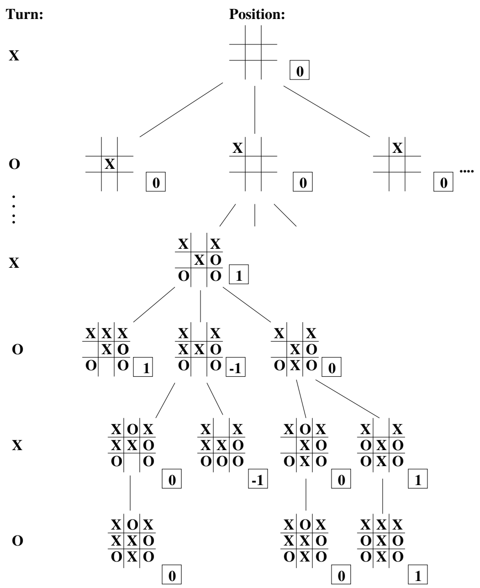
A backtrack search generates a *search tree* of increasingly complete partial solutions.

Let us illustrate this in the case of evaluating position values in a game of 3×3 noughts-and-crosses.

Associate to each incomplete position t in the game its *payoff value* for player X:

$$\text{payoff}(t) = \begin{cases} 1 & \text{if X has a winning strategy from } t, \\ -1 & \text{if O has a winning strategy from } t, \\ 0 & \text{if neither has a winning strategy from } t. \end{cases}$$

The complete annotated search, or *game tree* of this game is illustrated on the next slide.



The Minimax Rule

The payoff values for all positions can be computed by augmenting a backtrack search of the game tree with computations according to the following *minimax rules:*

Position type	*Payoff value*
final (complete)	can be determined directly
X moves	payoff = max {payoff values of immed. extensions}
O moves	payoff = min {payoff values of immed. extensions}

Bounded Depth Search

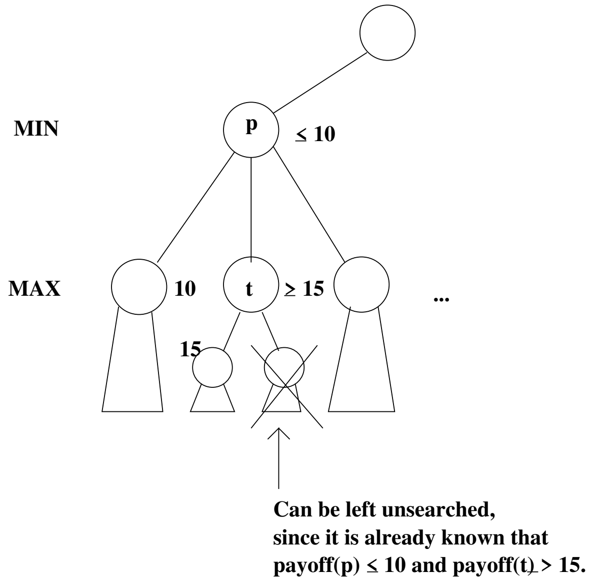
Game trees in realistic games are usually evaluated only to some predetermined *lookup depth* k, at which some heuristic *evaluation function* eval(t) is applied to estimate the payoff values of the incomplete positions t.

function payoff (t:position; k:depth; m:{MIN, MAX}):
  if $k = 0$ or t is a final position **then**
    return eval(t)
  else
    if m = MAX **then** $v \leftarrow -\infty$ **else** $v \leftarrow \infty$;
    for all t's extensions s **do**
      if m = MAX **then**
        $v \leftarrow$ max(v, payoff(s, $k-1$, MIN))
      else
        $v \leftarrow$ min(v, payoff(s, $k-1$, MAX))
      end if;
    return v
  end if.

3.4 Alpha-Beta Pruning

The size of a search tree can often be considerably reduced by eliminating branches that cannot improve an already known solution. In the case of game trees this process is known as *alpha-beta pruning*.

During the backtrack search of the game tree, maintain at each node t an *intermediate payoff value*, which for MIN nodes is an upper bound on the eventual true payoff value, and for MAX nodes a lower bound. Then when it is clear that no further search below a given node t can improve the payoff value of its father, the remaining subtrees of t cab be pruned. (See next slide.)



function payoff$\alpha\beta$ (t, k, m, pv); $\qquad$ pv = father's intermediate payoff
    if $k = 0$ or t is a final position **then return** eval(t)
    else
        if m = MAX **then** $v \leftarrow -\infty$ **else** $v \leftarrow \infty$;
        for all t's extensions s **do**
            if m = MAX **then**
                $v \leftarrow$ max(v, payoff$\alpha\beta$(s, $k-1$, MIN, v));
                if $v \geq pv$ **then return** v
            else
                $v \leftarrow$ min(v, payoff$\alpha\beta$ (s, $k-1$, MAX, v));
                if $v \leq pv$ **then return** v
            end if;
        return v
    end if.

3.5 Branch-and-Bound Search (1/2)

Similar pruning techniques can greatly improve the efficiency of backtrack search in optimisation problems.

Consider e.g. the TSP problem and choose:

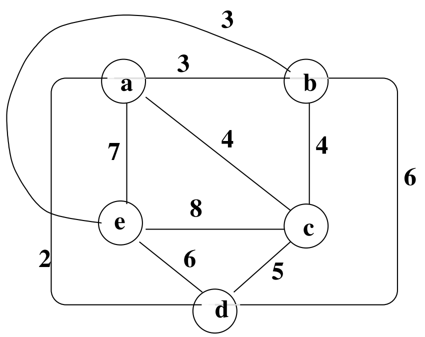
Partial solution: A set of edges (links) that have been decided to either include or exclude from the complete solution tour.
Bounding heuristic: Let the TSP instance under consideration be given by distance matrix $D = d_{ij}$. Then the following inequality holds for any complete tour π:

$$\begin{aligned} d(\pi) &= \frac{1}{2}\sum_{j}\{(d_{ij} + d_{jk}) \mid \text{at city } j \text{ tour } \pi \text{ uses links } ij \text{ and } jk\} \\ &\geq \frac{1}{2}\sum_{j}\min_{i,k}(d_{ij} + d_{jk}). \end{aligned}$$

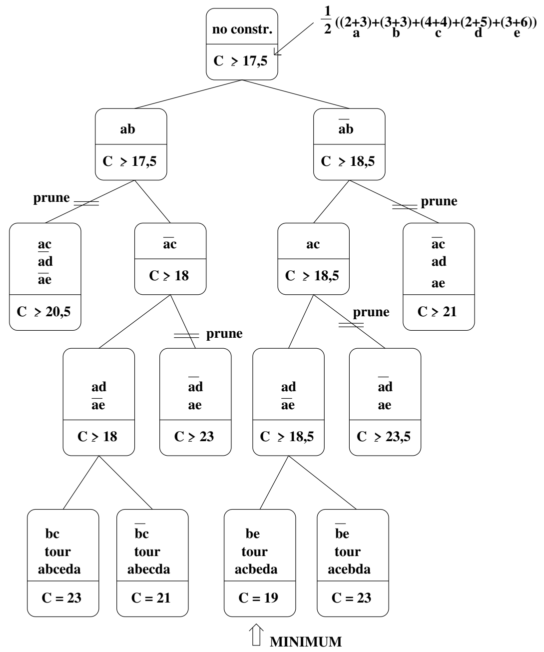
This estimate can be used to lower bound the length of tours achievable from any given partial solution, and prune the search tree correspondingly.

Branch-and-Bound Search (2/2)

Consider the following small TSP instance:



Using the above lower-bounding heuristic, the search tree for the minimum tour on this instance can be pruned as presented on the following slide.



3.6 The A* Algorithm

A* is basically a reformulation of the branch-and-bound search technique in terms of path search in graphs.

Given:

- search graph [neighbourhood structure] (X, N)
- *start node* $x_0 \in X$
- set of *goal nodes* $X^* \subseteq X$
- *edge costs* $c(x, x') \geq 0$ for $x \in X$, $x' \in N(x)$

Task: find a (minimum-cost) path from x_0 to some $x \in X^*$.

A*: Path Length Estimation

An important characteristic of A* is that the remaining distance from a node x to a goal node is estimated by some *heuristic* $h(x) \geq 0$.

As the algorithm visits a new node, it is placed in a set OPEN. Nodes in OPEN are selected for further exploration in increasing order of the *evaluation function*

$$f(x) = g(x) + h(x),$$

where $g(x) = \text{dist}(x_0, x)$ is the shortest presently known distance from the start node.

A heuristic $h(x)$ is *admissible*, if it underestimates the true remaining minimal distance $h^*(x)$, i.e. if for all $x \in X$:

$$h(x) \leq h^*(x) := \min_{x^* \in X^*} \text{dist}(x, x^*).$$

```
function A*(X, N, x0, c, h):
    place x0 in OPEN; set g(x0) = 0;
    while OPEN ≠ ∅ do
        choose some x ∈ OPEN for which f(x) is minimum;
        if x ∈ X* then return {found path to x};
        move x from OPEN to CLOSED;
        for all x' ∈ N(x) do
            if x' is not yet in OPEN or CLOSED then
                estimate h(x');
                compute f(x') = g(x') + h(x'),
                    where g(x') = g(x) + c(x, x');
                place x' in OPEN
            else {x' is already in OPEN or CLOSED}
                recompute f(x') = g(x') + h(x');
                if x' was in CLOSED and its f-value decreased then
                    move x' from CLOSED to OPEN
    end while;
    return fail {no path to goal found}.
```


A*: Convergence

A basic property of the A* algorithm is the following:

Theorem. Assume that the heuristic h is admissible. If the graph (X, N) is finite, and some path from x_0 to X^* exists, then A* returns one with a minimum cost.

Note 1: This result holds even for infinite search graphs satisfying some structural conditions. (Every node has only finitely many neighbours and all infinite paths have infinite cost.)

Note 2: Convergence of the algorithm can be guaranteed also for nonadmissible heuristics, but very little can be said about the cost of the paths returned in that case.

Note 3: The special case $h(x) \equiv 0$ reduces to the well-known Dijkstra's algorithm for shortest paths in graphs.

A*: Examples

In these two examples of A* search in graphs with obstacles, the heuristic $h(x)$ is taken to be the Manhattan (square-block) distance from a node x to the goal node x^* when the obstacles are ignored. The white nodes are in OPEN and the black nodes in CLOSED when the algorithm terminates.