Informed Search

Ian Sanders

Second Semester, 2024

Where to now?

- Previously *uninformed strategy* for node expansion
  - Shallowest
  - Deepest

Where to now?

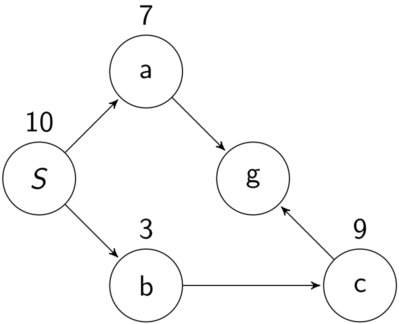
- Previously *uninformed strategy* for node expansion
  - Shallowest
  - Deepest – depth first search
  - Smallest total cost from root
- Now
  - What if we know something about the problem?
  - How can we incorporate knowledge?
  - Task-specific expansion strategy?
  - What if we have an adversary?
    - Can't just find a goal
    - Opponent can prevent us from doing so!

From UCS to heuristics

- Recall: UCS is like BFS with priority queue
- Nodes ordered by priority from smallest to largest
- Priority is cost estimate $f(n)$
- In UCS, $f(n) = g(n)$, where g is cost of reaching n from root
- Now we include a *Heuristic function*
- $h(n) =$ estimate of cost from n to goal
- $g(n)$ is backward cost (start to node), h is (estimated) forward cost (node to goal)
- So $f(n) = g(n) + h(n)$
- This leads to the idea of *best first search* – a generalisation of "normal" tree and graph searches.

Greedy best first search

- UCS is $f(n) = g(n)$ – no heuristic
- GBFS is $f(n) = h(n)$ – $h(n)$ is the heuristic, the estimate of the future cost to the goal
  Uses the heuristic in the hope that it is a good guide to the solution.



GBFS

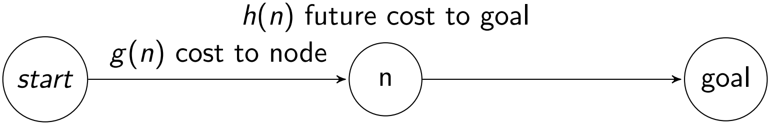
- Order nodes in the priority queue by estimated cost to goal
- Where does this estimate come from?
- Domain knowledge!
- For example, in choosing the shortest path from one town to another in a graph of towns, we could choose the *direct distance* ("as the crow flies") between two towns as the heuristic to use.

GBFS properties

- Incomplete for tree search (i.e. repeated nodes allowed)
- Complete for graph search in finite space
- Space and time complexity are both $O(b^m)$ for max search depth
- But can be reduced substantially depending on the heuristic and problem

A^* search

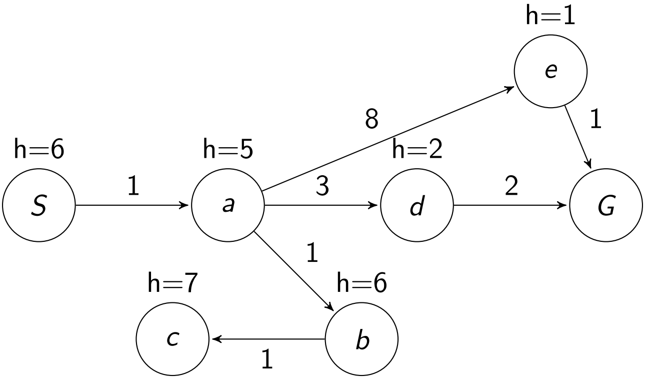
- A^* search is the most well known of all *best first searches*
- Recall
  UCS: $f(n) = g(n)$
  GBFS: $f(n) = h(n)$
- For A^* Search we use $g(n)$ the known path cost and $h(n)$ which is an estimate of *future cost*.



- Implement A* as UCS with different priority!

A* vs UCS vs GBFS

- Uniform-cost orders by path cost, or backward cost $g(n)$
- Greedy orders by goal proximity, or forward cost $h(n)$
- A* Search orders by the sum: $f(n) = g(n) + h(n)$

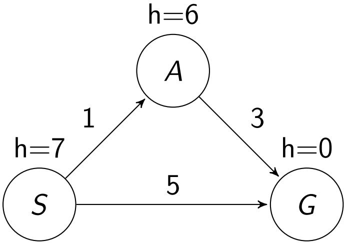


A* vs UCS vs GBFS

Search	expand	frontier
UCS	S	$a,\ g = 1$
GBFS	S	$a,\ h = 6$
A*	S	$a,\ f = 1 + 6 = 7$
UCS	a	$d,\ g = 4$
		$b,\ g = 5$
		$e,\ g = 9$
GBFS	a	$e,\ h = 1$
		$d,\ h = 3$
		$b,\ h = 5$
A*	a	$d,\ f = 4 + 2 = 6$
		$b,\ f = 2 + 6 = 8$
		$e,\ f = 9 + 1 = 10$

Optimality of A*

- Depends on heuristic



- Expand S
  To G – f is 5
  To A – f is 7
- First choice is $S \rightarrow G$
  Not the the best of options
  A better heuristic should have made A a better option.

Heuristics in A*

- h admissible if $hn \leq TrueFutureCost(n)$
  We never overestimate the cost!
- $f(n) = g(n) + h(n)$ means we never overestimate cost from start to goal through n
- Optimistic: estimates cost as less than it actually is
- h is consistent if $h(n) \leq c(n, a, n') + h(n')$
  n' is a successor of n, a is the action, c step cost to reach n' from n
  The triangle inquality
- All consistent heuristics are admissible
- A* tree search is optimal if h is admissible
- A* graph search is optimal if h is consistent

Properties of A*

- A* is complete, optimal
- A* is optimally efficient w.r.t. heuristic
  No other algorithm will expand fewer nodes
- Time complexity is $O(b\epsilon d)$
  ϵ is relative error of heuristic, d is solution depth
- Alternate view, $O({b^*}^d)$ where b^* is effective branching factor
- A* doesn't need to consider certain nodes (prunes) and so branching factor is reduced!
- Since exponential, memory is biggest issue
- Can do iterative deepening equivalent (IDA*)

Creating heuristics

- Main challenge for A*: how to make good heuristics that are admissible/consistent?
- For navigation, straight-line path is good
- Can never be faster than that!
- Could learn heuristics from data (e.g. machine learning/precomputed database lookups)
- Use relaxations – solve an easier, less constrained version of a problem
- E.g. Relaxed version of a maze – remove all the walls
- Tradeoff between quality of estimate and work per node
- Closer heuristic is to true cost, the fewer nodes are expanded...

Adversarial search

- So far, aim was to reach goal with min cost
- But what if another agent is trying to stop us?
- Must take into account their actions

Games are big

The size of the state space in some games....

- Tic-tac-toe $\approx 10^3$
- Connect Four $\approx 10^3$
- English draughts $\approx 10^{23}$
- Othello $\approx 10^{28}$
- Chess $\approx 10^{44}$
- Shogi $\approx 10^{71}$
- $\#$ atoms in observable universe $\approx 10^{82}$
- Twixt $\approx 10^{140}$
- Go (19x19 board) $\approx 10^{170}$

Games are hard

- NP-Hard or harder
  If a problem is NP-hard, then it is at least as difficult to solve as the problems in NP
  A Problem X is NP-Hard if there is an NP-Complete problem Y, such that Y is reducible to X in polynomial time.
  NP-Hard problems are as hard as NP-Complete problems.
  NP-Hard Problem need not be in NP class.
  NP-Complete is a subset of NP-Hard

Zero-sum games

- Assume two players, competitive
- Player 1 wins, player 2 loses and vice versa
- Game is defined by:
  - Initial state
  - $Player(s)$: whose turn it is
  - $Actions(s)$: available actions
  - $Result(s, a)$: successor function or transition model
  - $Terminal(s)$: is the game over/state terminal
  - $Utility(s, p)$: the value for the game ending in s for player p
- Zero sum game: sum of utilities for all players is constant: e.g. Win = +1, Draw = 0, Loss = -1

Two player, zero-sum

- Two players, MAX and MIN
- We are MAX, try to maximise utility
- Opponent is MIN, tries to minimise utility
- Denote $V(s)$ as utility at a given state
- Utility is known at terminal states
- Start at root node, expand tree
- Players alternate turns
- At level 0, us to play. Level 1, them to play, etc.
  Each level is called a *ply*.
- We want to compute optimal play, assuming our opponent is also optimal

Calculating minimax

- Want to calculate $V(s)$ for all s
- If s is terminal, use utility function directly
- else
  - if player to play is MAX:
  - Value is best maximising value at state
  - else player to play is MIN:
  - Value is best minimising value at state

Calculating minimax

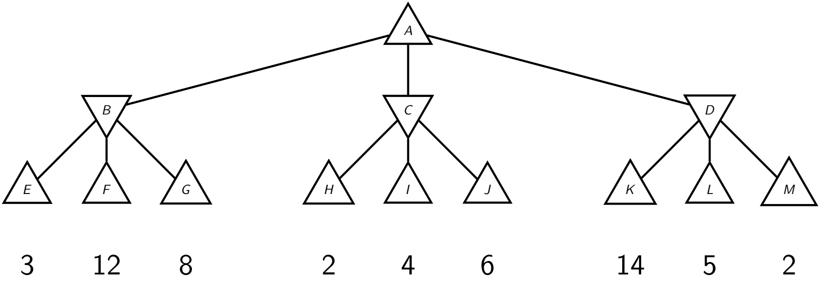
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

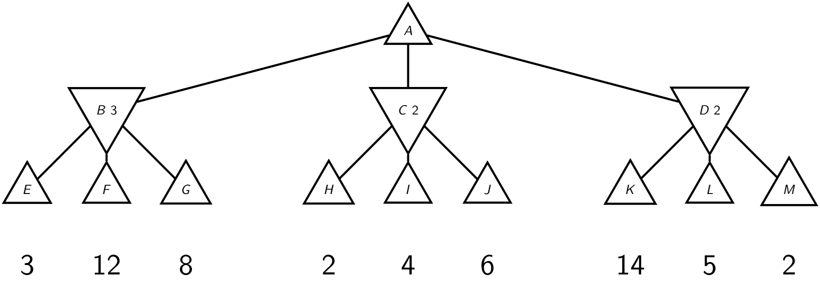
```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

$$
\begin{aligned}
&MiniMax(s) \\
&= Utility(s), If\ Terminal - Test(s) \\
&= max_{a \in Actions(s)} MiniMax(Result(s, a)), If\ Player(s) = Max \\
&= min_{a \in Actions(s)} MiniMax(Result(s, a)), If\ Player(s) = Min
\end{aligned}
$$

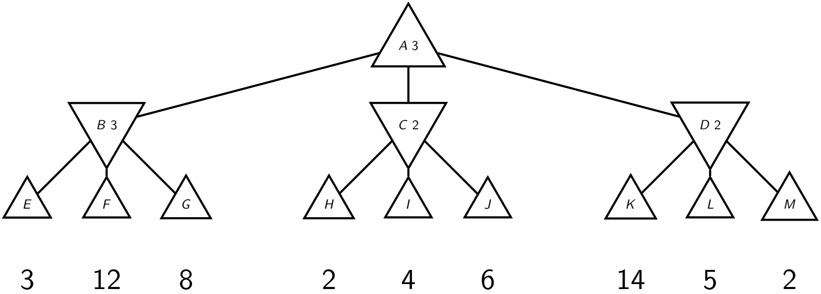
Example



Example



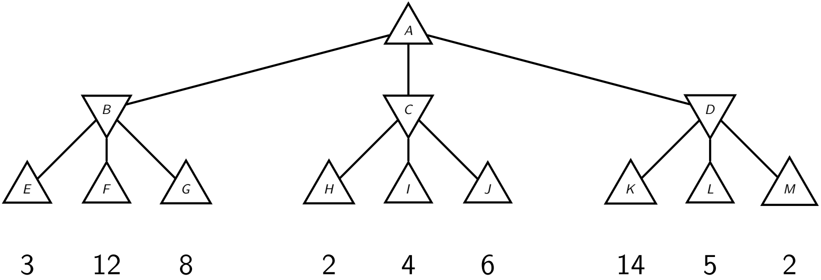
Example



Minimax properties

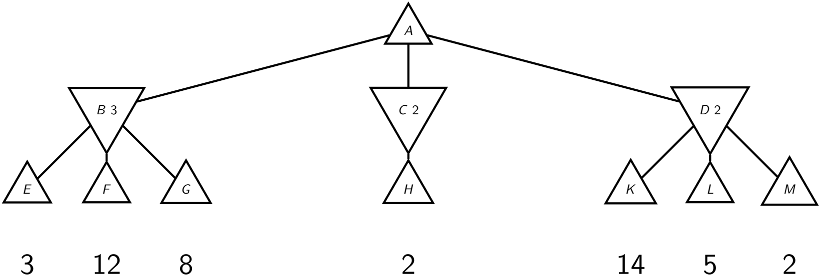
- Like DFS:
- Time: $O(b^m)$
- Space: $O(bm)$
- But instead of searching for single goal, we need to *exhaustively* try everything!
  And we only get value at leaf nodes
- Chess, for e.g., $b \approx 20$, $m \approx 70$
  Exact solution *infeasible*
  So what do we do?

Pruning



Max to play.

Pruning

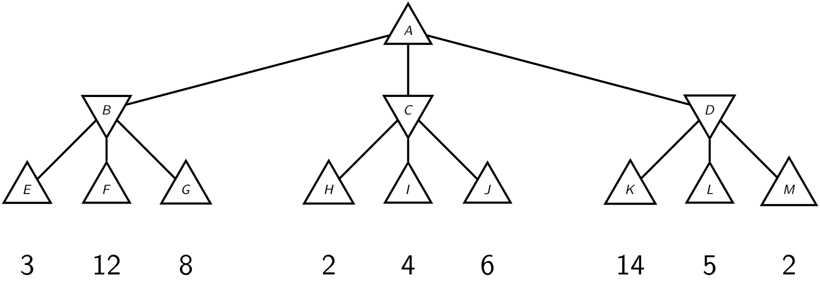


Max gets at least 3 – from the minimum of the left subtree
Min would consider H from this can get 2 for node C but this would not be chosen anyway as Max already has 3
If I and/or J where greater than 2 then Min would not choose them.
If I and/or J where less than 2 then Min could choose them but it would make no difference up the tree
I and/or J can be pruned!

Where to now?

- $\alpha\beta$-pruning
- α – minimum score MAX is guaranteed of
- β – maximum score MIN is guaranteed of
- If at a given node, $\beta < \alpha$
- Then MIN can guarantee a score that makes MAX sad
- So MAX will never go down this road
- No need to expand rest of node's children!
- Symmetric argument for other way around
- If children are expanded in optimal order, complexity is halved: $O(b^{m/2})$

Pruning



$\alpha\beta$ Search Algorithm

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

function MAX-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

function MIN-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a), α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

Tracing $\alpha\beta$ pruning

1. A: max – $v = -\infty, \alpha = -\infty, \beta = \infty$ expand B
2. B: min – $v = \infty, \alpha = -\infty, \beta = \infty$, expand E
3. E: max – terminal state true, returns utility value, ie returns 3
4. B: min – was $v = \infty, \alpha = -\infty, \beta = \infty$
   v min of v and returned value
   $v = MIN(\infty, 3) = 3$
   v *not* $\leq \alpha$ so do not return now
   $v = \cancel{\infty}\, 3, \alpha = -\infty, \beta = \cancel{\infty}\, 3$, expand F
5. F: max – terminal state true, returns utility value, ie returns 12
6. B: min – $v = 3, \alpha = -\infty, \beta = 3$, expand G
7. F: max – terminal state true, returns utility value, ie returns 8
8. B: min – $v = 3, \alpha = -\infty, \beta = 3$, returns $v = 3$ to A

9. A: max – $v = -\infty, \alpha = -\infty, \beta = \infty$
   v = MAX(v,3) = 3
   v not $\geq \beta$ so do not return
   α =MAX(α, v) = 3
   $v = \cancel{-\infty}\, 3, \alpha = \cancel{-\infty}\, 3, \beta = \infty$
   Expand C
10. C: min – $v = \infty, \alpha = 3, \beta = \infty$, expand H
11. H: max – terminal state true, returns utility value, ie returns 2
12. C: min
    before expanding H – $v = \infty, \alpha = 3, \beta = \infty$
    v min of v and returned value
    $v = MIN(\infty, 2) = 2$
    now $v \leq \alpha$ so return 2 to A
    This effectively prunes nodes I and J from the search tree.

13. A: max – $v = 3, \alpha = 3, \beta = \infty$ when calling C
    $v = MAX(v, 2) = 2$
    v not $\geq \beta$
    $\alpha = MAX(\alpha, v) = MAX(3, 2) = 3$
    So $v = 3, \alpha = 3, \beta = \infty$
    Expand D
14. D: min – $v = \infty$, $\alpha = 3$, $\beta = \infty$, Expand K
15. Continue as an exercise.

Depth-limited search

- Even with pruning, can't reach leaf nodes in real games
- So must limit depth of search
- Must replace utility function with estimate (like heuristic in A*)
- No longer optimal
- More plies = better performance
- Given time budget, use IDS!

Evaluation functions

- Estimate of utility of non-terminal state
- Ideally: want actual minimax value of state
- But this is unknown
- In practice: use domain knowledge
- Tradeoff between complexity vs depth
- More complex evaluation function may be more accurate, but take longer to compute therefore less time to search deeper
- Stockfish: fast eval function, huge depth
- Komodo: slow, complex eval function, less depth

Other improvements

- Base algorithm of $\alpha\beta$ + IDS + eval function
- Transposition tables: stored previous states and their evals
- Aspiration windows: pretend that the $\alpha\beta$ window is smaller than it is
- Evaluation functions optimised from data (machine learning, etc.)
- Move ordering: try certain classes of moves first (e.g. captures, then regular moves)

Summary

- For single-agent search:
- Uninformed node expansion (DFS/BFS/UCS)
- Informed with heuristics (GBFS/A*)
- A* optimal, but need admissible/consistent heuristics
- Heuristics from problem relaxation
- Adversarial search:
- Minimax for optimal play
- Can use pruning to reduce nodes
- In practice, must cut-off depth
- Use evaluation function

Primary focus of AI for 60 years!