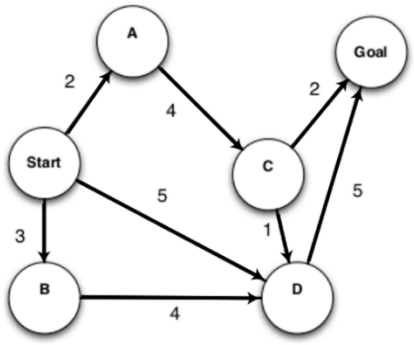
CS 461 Homework 1 Search Algorithms
Görkem Kadir Solun 22003214

1 Uninformed Search

Depth-first search

Fringe
~~S~~
~~S > A~~
S > B
S > D
~~S > A > C~~
~~S > A > C > D~~
S > A > C > G
~~S > A > C > D > G~~



For each of the following graph search strategies, work out the order in which states are expanded, as well as the path returned by graph search. In all cases, assume ties resolve in such a way that states with earlier alphabetical order are expanded first. Remember that in graph search, a state is expanded only once.

Start > A > C > D > Goal

Breadth-first search

Fringe
~~S~~
~~S > A~~
~~S > B~~
~~S > D~~
~~S > A > C~~
S > B > D
~~S > D > G~~
S > A > C > D
S > A > C > G

Start > D > Goal

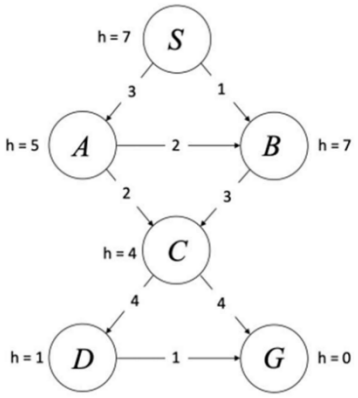
Uniform cost search

Fringe	Cost
~~S~~	0
~~S > A~~	2
~~S > B~~	3
~~S > D~~	5
~~S > A > C~~	6
S > B > D	7
S > D > G	10
S > A > C > D	7
S > A > C > G	8

UCS gives S > A > C > G with a cost of 8.

2 Informed Search

Select all boxes that describe the given heuristic values. If you think they are not admissible and/or consistent, give a counterexample to show it.



We will investigate various informed search algorithms for the following graph. Edges are labeled with their costs, and heuristic values h for states are labeled next to the states. S is the start state, and G is the goal state. In all search algorithms, assume ties are broken in alphabetical order.

Consistent and admissible (Consistency implies admissibility)

No node/edge violates the admissibility condition $0 \leq h(n) \leq c(n)$.

No node/edge violates the consistency condition $h(prev) \leq c(prev, next) + h(next)$

Prev is the previous node, and next is the successor.

Given the above heuristics, what is the order in which the states will be expanded, assuming we run a greedy graph search with the heuristic values provided?

Index	1	2	3	4	5	Not Expanded
S	X					
A		X				
B						X
C			X			
D						X
G				X		

Assuming we run a greedy graph search with the heuristic values provided, what path is returned?

S > A > C > G

Given the above heuristics, what is the order in which the states are going to be expanded, assuming we run an A* graph search with the heuristic values provided?

Index	1	2	3	4	5	Not Expanded
S	X					
A		X				
B			X			
C				X		
D						X
G					X	

Assuming we run an A* graph search with the heuristic values provided, what path is returned?

Fringe	Cost (g + h)
~~S~~	0
~~S > A~~	3+5
~~S > B~~	1+7

S > A > C	5+4
~~S > B > C~~	4+4
S > B > C > D	8+1
~~S > B > C > G~~	8+0

A* returns S>B>C>G

3 Iterative Deepening A*

Consider a variant of iterative deepening called iterative deepening A*, where instead of limiting the depth-first search by depth as in standard iterative deepening search, we can limit the depth-first search by the f value as defined in the A* search. As a reminder, f[node] = g[node]+h[node] where g[node] is the cost of the path from the start state, and h[node] is a heuristic value estimating the cost to the closest goal state.

Assuming there are no ties in the f value between nodes, which of the following statements about the number of nodes that iterative deepening A* expands is True? If the same node is expanded multiple times, count all the times it is expanded. If none of the options are correct, mark None of the above. In any case, explain your reasoning below.

[TRUE] The number of times that iterative deepening A* expands a node is greater than or equal to the number of times A* will expand a node.

[FALSE] The number of times that iterative deepening A* expands a node is less than or equal to the number of times A* will expand a node.

[FALSE] We don't know if the number of times iterative deepening A* expands a node is more or less than the number of times A* will expand a node.

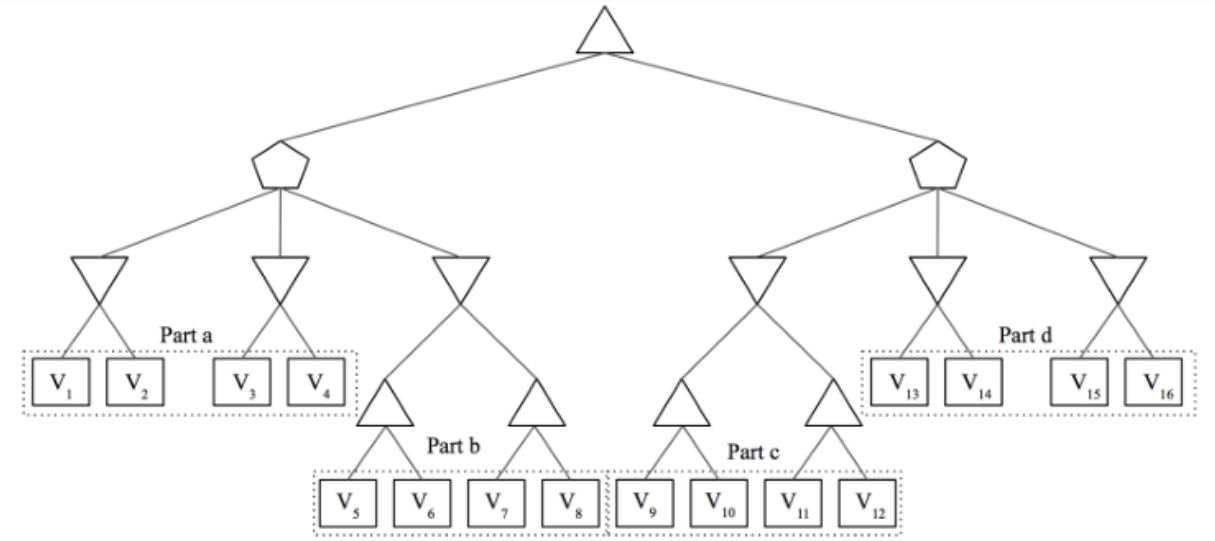
[FALSE] None of the above

A* uses a priority queue to keep track of nodes that need to be expanded. It keeps the best path to each node and avoids re-expanding nodes if a better path is not found. Nodes are typically expanded once, as A* can detect and discard suboptimal paths.

IDA* performs depth-first searches with increasing f-value thresholds. Due to its depth-first nature, it does not retain all generated nodes in memory. Nodes can be expanded multiple times within the same iteration and across different iterations with higher thresholds.

The number of times IDA* expands nodes is greater than or equal to the number of times A* expands nodes because IDA* may revisit nodes multiple times, whereas A* efficiently avoids unnecessary re-expansions.

4 Adversarial Search



Consider a game tree with 3 kinds of nodes: maximizers (up triangles), minimizers (down triangles), and ”medianizers” (pentagons). Medianizers select the median of their children. We say that a node is prunable if there is any possible configuration of utilities where that node’s utility is guaranteed not to affect the root utility; the configuration for prunability does not have to be the same for two different nodes. Notice that alpha-beta pruning is not applicable here; however, we can find conditions where a node can be pruned with a similar reasoning. You may assume leaf nodes are evaluated left-to-right, and Vi’s are unique.

Determine which nodes labeled ”Part a” are prunable, and explain your reasoning below.

Node	Prunable	Not prunable
V1		X
V2		X
V3		X
V4		X

Nothing is prunable since we must determine the median and the values may change the median.

Determine which nodes labeled as ”Part b” are prunable, and explain your reasoning below.

Node	Prunable	Not prunable
V5		X
V6	X	
V7	X	
V8	X	

There are two conditions for pruning:

Pruning V7 and V8: If the minimizers from part (a) are greater than the maximizers produced by V5 and V6, then V7 and V8 can be pruned. This is because the minimizer derived from the maximizers of V5 and V6 will be smaller than those of V7 and V8. Since the minimizer from part (a) is already known, the outcome can be determined without further evaluation of V7 and V8.

Pruning V6: V6 can be pruned if the minimizers from part (a) are smaller than V5. In this case, the maximizer produced by V5 and V6 will be larger than V5, making V6 irrelevant. However, V7 must still be checked in this scenario, as its minimizer could influence the final medianizer and potentially alter the result.

Determine which nodes labeled as ”Part c” are prunable, and explain your reasoning below.

Node	Prunable	Not prunable
V9		X
V10		X
V11	X	
V12	X	

V11 and V12 can be pruned if the values of V9 and V10 are smaller than the maximum value obtained from the median of the left branch. This is because the minimum value derived from the maximum of V9 and V10 will only decrease, making the right branches irrelevant.

The reasoning behind this is that if V9 and V10 are smaller than the minimum derived from the maximum of their own values, then the result will be less than the left branch’s median.

In the case where one right branch has a smaller value while the other has a larger one, the minimum from the maximum of V9 and V10 will still be smaller, so it won’t be chosen, as it is already less than the median of the left branch. If both values from V9 and V10 are larger, then the minimum from V9 and V10 will either be smaller or equal to the current value, which also poses no issue.

This pruning ensures that unnecessary computations are avoided, as the right branch outcomes are irrelevant when compared to the left branch's median.

Determine which nodes labeled as ”Part d” are prunable, and explain your reasoning below.

Node	Prunable	Not prunable
V13		X
V14	X	
V15	X	
V16	X	

V16, V15, and V14 can be pruned if the left medianizer has a higher value than the left child of the right medianizer and V13. Having lower values in the left child of the right medianizer and V13 guarantees the right medianizer to take a lower value than the left medianizer. As we have a maximizer in the root, this makes V16, V15, and V14 prunable.

5 Monte Carlo Tree Search

Explain the critical steps of the vanilla Monte Carlo Tree Search (MCTS) algorithm. How does each step contribute to the overall search process?

The vanilla MCTS algorithm consists of four key steps, each contributing uniquely to building an optimal search tree for making informed decisions: Selection, Expansion, Simulation (Rollout), and Backpropagation.

Selection

The selection phase involves navigating from the root node (the current game state) down the tree to a leaf node (a node that is not fully expanded or is terminal) using a tree policy. Starting at the root, the algorithm recursively selects child nodes based on a selection policy, often the highest UCB value. The formula balances between exploiting well-performing actions and exploring less-visited ones. Using the UCB formula, the selection step balances between exploring nodes with few visits (exploration) and nodes with high average reward (exploitation). It prioritizes promising areas of the search space, ensuring that the algorithm efficiently uses computational resources to explore the most relevant parts of the tree.

UCB heuristics

- UCB1 (upper confidence bound) formula combines “promising” and “uncertain”:

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

- **Intrinsic** utility for exploration (in addition to the **extrinsic** task utility)
- $N(n)$ = number of rollouts from node n
- $U(n)$ = total utility of rollouts (e.g., # wins) for player of Parent(n)

Figure 1: UCB formula taken from slide 4-2

Expansion

Upon reaching a leaf node not in a terminal state, the algorithm expands the tree by adding one or more child nodes representing possible future moves. Typically, the algorithm adds a single child node per iteration to keep the tree growth manageable. Each new node represents a potential action from the parent node's state. Expansion allows the algorithm to consider new actions and states that have yet to be explored. The algorithm incrementally builds a more comprehensive search space representation by gradually expanding the tree.

Simulation (Rollout)

The simulation phase involves playing out the game from the newly expanded node to a terminal state (end of the game) using a default rollout policy. The algorithm simulates a sequence of moves by selecting actions until the game ends. Alternatively, simple heuristics can guide the simulation to make it more informed than random play. Simulation provides a statistical estimate of the potential success of the new node. The outcome of the simulation influences the perceived value of the moves leading up to it, affecting future selection and expansion phases.

Backpropagation

Backpropagation updates the statistics of all nodes along the path from the expanded node back to the root based on the simulation result. The algorithm increments the visit count for each node in the path and updates the total reward (win/loss). The updates proceed from the leaf node to the root, ensuring all relevant nodes are informed of the simulation outcome. The algorithm reinforces the paths that lead to successful outcomes by updating the nodes with the results. The updated statistics directly influence the selection phase in future iterations, improving the algorithm's decision-making over time.

The interplay of these four steps enables MCTS to search through a vast decision space efficiently. Each iteration of the four steps refines the search tree, making the algorithm progressively better at estimating the value of different actions. MCTS balances the need to explore uncharted territories with the exploitation of known rewarding paths. Given sufficient time and iterations, MCTS converges towards the optimal decision, making it a powerful tool for complex decision-making tasks.

Describe how the Upper Confidence Bound (UCB) formula is used during the selection phase of MCTS. What is the intuition behind balancing exploration and exploitation? What happens if you set the constant c too high or too low?

As described above and given in Figure 1, the Upper Confidence Bound (UCB) in MCTS is a critical component that guides the selection phase of the algorithm. It helps make informed decisions about which node to explore next by balancing two competing objectives: exploration (sampling less-visited nodes to discover their potential) and exploitation (focusing on nodes known to yield high rewards).

Traverse the tree from the root node to a leaf node by selecting child nodes at each level. At each decision node/point, the algorithm uses the UCB formula to evaluate and choose the best child node to proceed with.

Exploitation

It is leveraging accumulated knowledge to choose the best-known option. The term represents the average reward of the node. Encourages the algorithm to select nodes that have historically yielded high rewards.

$$\frac{U(n)}{N(n)}$$

Exploration

It is investigating less-visited nodes to discover potentially better options. The term increases for nodes with fewer visits N(n) and as the total number of simulations Parent(n) increases. Promotes the selection of nodes that have yet to be thoroughly explored, preventing the algorithm from getting stuck in suboptimal regions.

$$C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

Balancing

The algorithm must balance exploiting known high-reward paths and exploring new paths that might yield even better rewards. The exploration constant C adjusts the weight given to the term, effectively tuning the balance between exploration and exploitation. The algorithm might converge on suboptimal moves without sufficient exploration because it needs to explore alternatives adequately. Balancing ensures that computational resources are allocated to refining estimates of known good moves and discovering new, potentially better ones.

Setting C High

A high C value amplifies the exploration term, making it more influential in the UCB calculation. The algorithm will over-prioritize less-visited nodes, even with lower average rewards.

Excessive exploration can lead to spending too much time on suboptimal or irrelevant nodes. The algorithm may take longer to identify the best moves because it diverts attention to underperforming nodes.

Setting C Low

A low C value diminishes the exploration term's influence, making the average reward the primary factor. The algorithm will heavily favor nodes with the highest average rewards so far. The algorithm may overlook better moves that are less explored, potentially settling for suboptimal strategies. A lack of exploration can make the algorithm less adaptable to environmental changes or uncertainties.

How do progressive widening and double progressive widening work? What types of problems would you expect progressive widening to be more effective than a standard MCTS approach? Provide an example problem.

When the action space is huge, expanding every potential child node from a parent node is often impractical. Progressive widening (PW) addresses this by selectively growing the tree, choosing a limited number of actions $\theta_1 N(n)^{\theta_2}$. Double progressive widening (DPW) extends this by applying the idea to actions and the state space. A new state is sampled if fewer than $\theta_3 N(n)^{\theta_4}$ different states have already been explored. These techniques are beneficial when considering all possible actions or states that are computationally expensive or infeasible. Unlike traditional MCTS, which uniformly expands child nodes, PW and DPW focus on the most promising actions. For instance, an exhaustive search would overwhelm computational resources in complex games that consider many actions where numerous variables exist. PW and DPW allow the system to explore the most relevant options efficiently.

When are UCB and progressive widening useful? When can it be more beneficial to not use UCB and progressive widening?

UCB is helpful in situations where there is a need to balance exploration and exploitation. It is particularly effective in problems where you must maximize a reward over time, such as in MCTS in games like chess.

Progressive Widening is useful when the action or state space is vast, and expanding every possible option would be computationally expensive or infeasible. Instead of developing all possible actions, progressive widening selects a subset based on the times the node has been visited. It is particularly effective in domains with a vast number of possibilities.

When can it be more beneficial not to use UCB and progressive widening?

If the action space is small, expanding all actions or states might not be computationally expensive, so progressive widening may not be necessary. Similarly, UCB's exploration-exploitation trade-off might not add much value in small-scale problems where all options can be evaluated quickly and an exhaustive search is feasible.

In environments where the action space is straightforward or where the dynamics of the problem are well-understood, more basic search strategies without UCB or progressive widening can suffice.

In deterministic environments with low variability, where actions produce predictable results, UCB's exploration component may lead to unnecessary exploration. In such cases, it may be more beneficial to exploit known good actions consistently without the need for UCB's balance between exploration and exploitation.