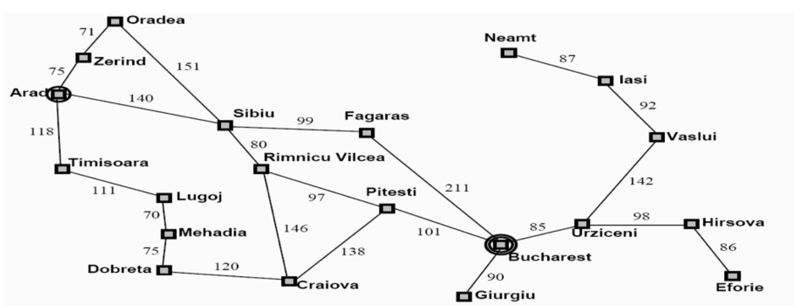
CSE 3401: Intro to AI & LP
Uninformed Search II

- Required Readings: R & N Chapter 3, Sec. 1-4.



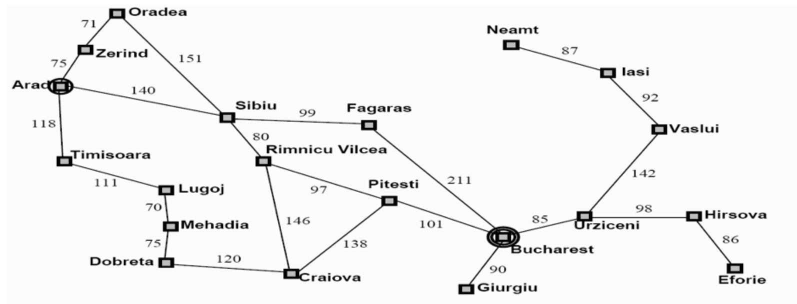
{Arad},

{Zerind, Timisoara, Sibiu},

{Zerind, Timisoara, Arad, Oradea, Fagaras, RimnicuVilcea },

{Zerind, Timisoara, Arad, Oradea, Sibiu, Bucharest, RimnicuVilcea },

Solution: Arad -> Sibiu -> Fagaras -> Bucharest
Cost: 140+99+211 = 450



{Arad},

{Zerind, Timisoara, Sibiu},

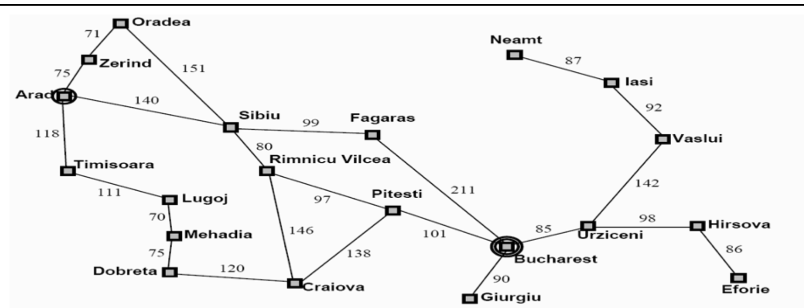
{Zerind, Timisoara, Arad, Oradea, Fagaras, RimnicuVilcea},

{Zerind, Timisoara, Arad, Oradea, Sibiu, Pitesi, Craiova<via RimnicuVilcea>},

{Zerind, Timisoara, Arad, Oradea, Sibiu, Craiova<via Pitesi>, Bucharest, Craiova<via RimnicuVilcea>},

- Solution: Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest

  Cost: 140+80+97+101 = 418



{Arad<>},

{Zerind<Arad>, Timisoara<Arad>, Sibiu<Arad>},

{Zerind <Arad>, Timisoara <Arad>, Oradea <Sibiu;Arad>, Fagaras<Sibiu;Arad>, Arad<Sibiu;Arad>, RimnicuVilcea<Sibiu;Arad>},

{Zerind <Arad>, Timisoara <Arad>, Oradea <Sibiu;Arad>, Fagaras<Sibiu;Arad>, Zerind<Arad;Sibiu;Arad>, Timisoara<Arad;Sibiu;Arad>, Sibiu<Arad;Sibiu;Arad>, RimnicuVilcea<Sibiu;Arad>},

No solution found, search does not terminate because of cycles!

Selection Rule.

- The example shows that order states are selected from the frontier has a critical effect on the operation of the search.
  - Whether or not a solution is found
  - The cost of the solution found.
  - The time and space required by the search.

Critical Properties of Search.

- Completeness: will the search always find a solution of a solution exists?
- Optimality: will the search always find the least cost solution? (when actions have costs)
- Time complexity: what is the maximum number of nodes than can be expanded or generated?
- Space complexity: what is the maximum number of nodes that have to be stored in memory?

Uninformed Search Strategies

- These are strategies that adopt a fixed rule for selecting the next state to be expanded.
- The rule is always the same whatever the search problem being solved.
- These strategies do not take into account any domain specific information about the particular search problem.
- Popular uninformed search techniques:
  - Breadth-First, Uniform-Cost, Depth-First, Depth-Limited, and Iterative-Deepening search.

Selecting vs. Sorting

- A simple equivalence we will exploit:
  - Order the elements on the frontier.
  - Always select the first element.
- Any selection rule can be achieved by employing an appropriate ordering of the frontier set.

Breadth First.

- Place the successors of the current state at the end of the frontier, which then behaves as a FIFO queue.
- Example:
  - let the states be the positive integers {0,1,2,…}
  - let each state n have as successors n+1 and n+2
    - E.g. S(1) = {2, 3}; S(10) = {11, 12}
  - Start state 0
  - Goal state 5
  - [Draw search space graph]

Breadth First Example.

{0}
{1,2}
{2,2,3}
{2,3,3,4}
{3,3,4,3,4}
{3,4,3,4,4,5}
…

[Draw search tree]

Breadth First Properties

- Measuring time and space complexity.
  - let b be the maximum number of successors of any state.
  - let d be the number of actions in the shortest solution.

Breadth First Properties

- Completeness?
  - The length of the path from the initial state to the expanded state must increase monotonically.
    - we replace each expanded state with states on longer paths.
    - All shorter paths are expanded prior before any longer path.
  - Hence, eventually we must examine all paths of length d, and thus find the shortest solution.

Breadth First Properties

- Time Complexity?
  - # nodes generated at…
  - Level 0 (root): 1
  - Level 1: 1* b [each node has at most b successors]
  - Level 2: b* b = b^2
  - Level 3: b * b^2 = b^3 ....
  - Level d: b^d
  - Level d + 1: $b^{d+1} - b = b(b^d - 1)$ [when last node is successful]
  - **Total:** $1 + b + b^2 + b^3 + \ldots + b^{d-1} + b^d + b(b^d - 1) = O(b^{d+1})$
  - Exponential, so can only solve small instances

Breadth First Properties

- Space Complexity?
  - $O(b^{d+1})$: If goal node is last node at level d, all of the successors of the other nodes will be on the frontier when the goal node is expanded, i.e. $b(b^d - 1)$

Breadth First Properties

- Optimality?
  - Will find shortest path length solution
  - Least cost solution?
    - In general no!
    - Only if all step costs are equal

Breadth First Properties

- Space complexity is a real problem.
  - E.g., let b = 10, and say 1000 nodes can be expanded per second and each node requires 100 bytes of storage:

Depth	Nodes	Time	Memory
1	1	1 millisec.	100 bytes
6	10^6	18 mins.	111 MB
8	10^8	31 hrs.	11 GB

- Run out of space long before we run out of time in most applications.

Uniform Cost Search.

- Keep the frontier sorted in increasing cost of the path to a node; behaves like priority queue.
- Always expand the least cost node.
- Identical to Breadth First if each transition has the same cost.

- Example:
  - let the states be the positive integers {0,1,2,…}
  - let each state n have as successors n+1 and n+2
  - Say that the n+1 action has cost 2, while the n+2 action has cost 3.
  - [Draw search space graph]

Uniform Cost Search.

{0[0]}
{1[2],2[3]}
{2[3],2[4],3[5]}
{2[4],3[5],3[5],4[6]}
{3[5],3[5],4[6],3[6],4[7]}
…

Uniform-Cost Search

- Completeness?
  - Assume each transition has costs $\geq \epsilon > 0$ (otherwise can have in finite path with finite cost)
  - The previous argument used for breadth first search holds: the cost of the expanded state must increase monotonically.
  - The algorithm expands nodes in order of increasing path cost.

Uniform-Cost Search

- Time and Space Complexity?
  - $O(b^{C^*/\epsilon})$ where C* is the cost of the optimal solution.
    - Difficulty is that there may be many long paths with cost $\leq$ C*; Uniform-cost search must explore them all.

Uniform-Cost Search

- Optimality?
  - Finds optimal solution if each transition has cost $\geq \epsilon > 0$.
    - Explores paths in the search space in increasing order of cost. So must find minimum cost path to a goal before finding any higher costs paths.

Uniform-Cost Search. Proof of Optimality.

1. Let c(n) be the cost of the path to node n. If n2 is expanded after n1 then
   $c(n1) \leq c(n2)$.

Proof:

- If n2 was on the frontier when n1 was expanded, in which case $c(n2) \geq c(n1)$ else n1 would not have been selected for expansion.
- If n2 was added to the frontier when n1 was expanded, in which case $c(n2) \geq c(n1)$ since the path to n2 extends the path to n1.
- If n2 is a successor of a node n3 that was on the frontier or added when n1 was expanded, then $c(n2) > c(n3)$ and $c(n3) \geq c(n1)$ by the above arguments.

Uniform-Cost Search. Proof of Optimality.

2. When n is expanded every path with cost strictly less than c(n) has already been expanded (i.e., every node on it has been expanded).

Proof:

- Let <Start, n0, n1, …, nk> be a path with cost less than c(n). Let ni be the last node on this path that has been expanded. <Start, n0, n1, ni-1, ni, ni+1, …, nk>.
- ni+1 must be on the frontier, also c(ni+1) < c(n) since the cost of the entire path to nk is < c(n).
- But then uniform-cost would have expanded ni+1 not n!
- So every node on this path must already be expanded, i.e. this path has already been expanded. QED

Uniform-Cost Search. Proof of Optimality.

3. The first time uniform-cost expands a state, it has found the minimal cost path to it (it might later find other paths to the same state).

Proof:

- No cheaper path exists, else that path would have been expanded before.
- No cheaper path will be discovered later, as all those paths must be at least as expensive.
- So, when a goal state is expanded, the path to it must be optimal.

Depth First Search

- Place the successors of the current state at the front of the frontier.
- Frontier behaves like a stack.

Depth First Search Example

(applied to the example of Breadth First search)

{0}

{1,2}

{2,3,2}

{3,4,3,2}

{4,5,4,3,2}

{5,6,5,4,3,2}

…

[draw search tree]

Depth First Properties

- Completeness? No!
  - Infinite paths cause incompleteness! Typically come from cycles in search space.
  - If we prune paths with duplicate states, get completeness provided the search space is finite.
- Optimality? No!
  - Can find success along a longer branch!

Depth First Properties

- Time Complexity?
  - $O(b^m)$ where m is the length of the longest path in the state space.
  - Why? In worst case, expands

$1 + b + b^2 + \ldots + b^{m-1} + b^m = b^{m+1} - 1/b - 1 = O(b^m)$ nodes

  - Assumes no cycles.
  - Very bad if m is much larger than d, but if there are many solution paths it can be much faster than breadth first.

Depth First Backtrack Points

At each step, all nodes in the frontier (except the head) are backtrack points (see example and draw the tree for state-space).

These are all siblings of nodes on the current branch.

Depth First Properties

- Space Complexity?
  - O(bm), linear space!
    - Only explore a single path at a time.
    - The frontier only contains the deepest states on the current path along with the backtrack points.
  - Can reduce to O(m) if we generate siblings one at a time.

Depth Limited Search

- Breadth first has computational, especially, space problems. Depth first can run off down a very long (or infinite) path.
- Depth limited search.
  - Perform depth first search but only to a pre-specified depth limit L.
  - No node on a path that is more than L steps from the initial state is placed on the Frontier.
  - We “truncate” the search by looking only at paths of length L or less.
- Now infinite length paths are not a problem.
- But will only find a solution if a solution of length ≤ L exists.

Depth Limited Search

```
DLS(Frontier, Sucessors, Goal?)
   If Frontier is empty return failure

   Curr = select state from Frontier

   If(Goal?(Curr)) return Curr.

   If Depth(Curr) < L
       Frontier' = (Frontier - {Curr}) U Successors(Curr)

   Else
       Frontier' = Frontier - {Curr}
       CutOffOccured = TRUE.

   return DLS(Frontier', Successors, Goal?)
```

Iterative Deepening Search.

- Take the idea of depth limited search one step further.
- Starting at depth limit L = 0, we iteratively increase the depth limit, performing a depth limited search for each depth limit.
- Stop if no solution is found, or if the depth limited search failed without cutting off any nodes because of the depth limit.

Iterative Deepening Search Example

{0} [DL = 0]

{0} [DL = 1]
{1,2}
{2}

{0} [DL = 2]
{1,2}
{2,3,2}, {3,2}, {2}
{3, 4}, {4}

{0} [DL = 3]
{1,2}
{2,3,2}
{3,4,3,2}, {4,3,2}, {3,2}
{4,5,2}, {5, 2}
Success!

Iterative Deepening Search Properties

- Completeness?
  - Yes, if solution of length d exists, will the search will find it when L = d.
- Time Complexity?
  - At first glance, seems bad because nodes are expanded many times.

Iterative Deepening Search Properties

- Time Complexity
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

    [root expanded d+1 times, level 1 nodes expanded d times, …]
  - E.g. b=4, d=10
    - $(11)*4^0 + 10*4^1 + 9*4^2 + \ldots + 2*4^9 = 815{,}555$
    - $4^{10} = 1{,}048{,}576$
    - Most nodes lie on bottom layer.
    - In fact IDS can be more efficient than breadth first search: nodes at limit are not expanded. BFS must expand all nodes until it expands a goal node.

Iterative Deepening Search Properties

- Space Complexity
  - O(bd) Still linear!
- Optimal?
  - Will find shortest length solution which is optimal if costs are uniform.
  - If costs are not uniform, we can use a “cost” bound instead.
    - Only expand paths of cost less than the cost bound.
    - Keep track of the minimum cost unexpanded path in each depth first iteration, increase the cost bound to this on the next iteration.
    - This can be very expensive. Need as many iterations of the search as there are distinct path costs.

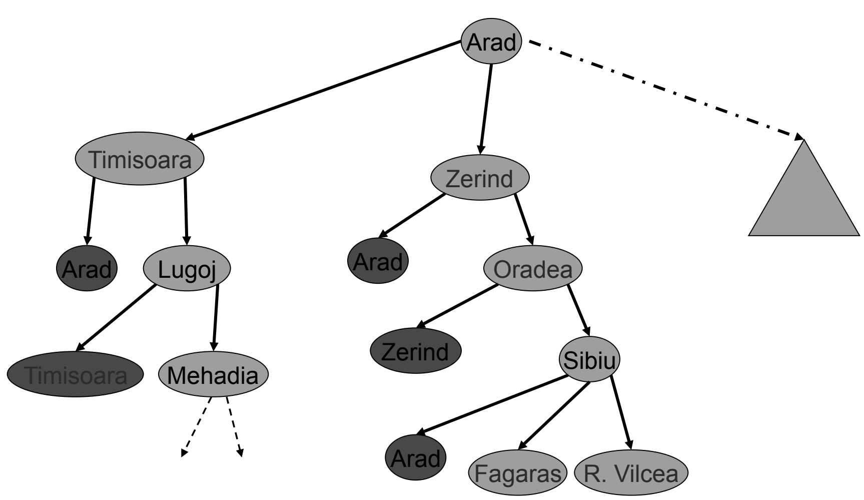
Iterative Deepening Search Properties

- Consider space with three paths of length 3, but each action having a distinct cost.

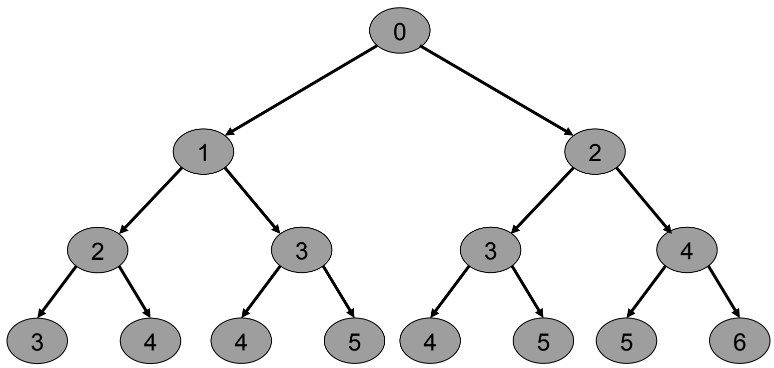
Cycle Checking

- Path checking
  - Paths are stored on the frontier (this allows us to output the solution path).
    - If <S,n_1,…,n_k> is a path to node n_k, and we expand n_k to obtain child c, we have
      - <S,n_1,…,n_k,c>
    - As the path to “c”.
  - Path checking:
    - Ensure that the state c is not equal to the state reached by any ancestor of c along this path.

Path Checking Example



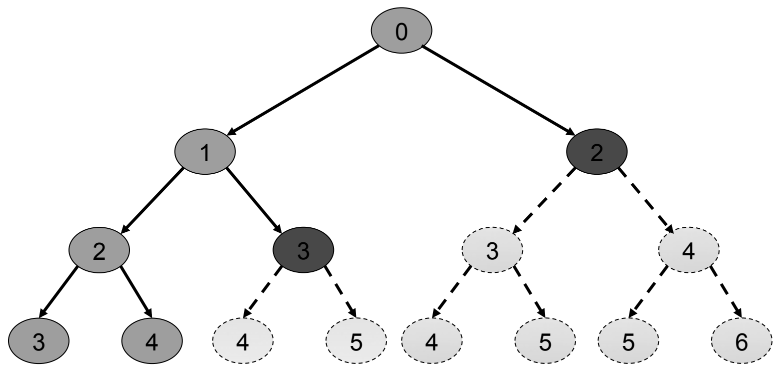
Path Checking Example



Cycle Checking

- Cycle Checking.
  - Keep track of all states previously expanded during the search.
  - When we expand n_k to obtain child c
    - ensure that c is not equal to any previously expanded state.
  - This is called cycle checking, or multiple path checking.
  - Why can't we utilize this technique with depth-first search?
    - If we use cycle checking in depth-first search what happens to space complexity.

Cycle Checking Example



Cycle Checking

- High space complexity, only useful with breadth first search.
- There is an additional issue when we are looking for an optimal solution
  - With uniform-cost search, we still find an optimal solution
    - The first time uniform-cost expands a state it has found the minimal cost path to it.
  - This means that the nodes rejected by cycle checking can't have better paths.
  - We will see later that we don't always have this property when we do heuristic search.