Introduction to Artificial Intelligence

Uninformed Search

Andres Mendez-Vazquez

January 14, 2020

Outline

1. Motivation
   - Mimicking the way Human Solve Problems
   - What is Search?
2. First Idea, State Space Problem
   - Introduction
   - Better Representation
     - Example
   - Solution Definition
   - Weighted State Space Problem
   - Evaluation of Search Strategies
   - Sparse Representation of Graphs
3. Uninformed Graph Search Algorithms
   - Implicit State Space Graph
   - Back to Implicit State Space Definition
   - Basic Functions
   - Depth-First Search
   - Breadth-First Search
   - Combining DFS and BFS
   - We Have the Results of Solving a Maze
4. Different ways of doing Stuff
   - What happened when you have weights?
   - What to do with negative weights?
   - Implicit Bellman-Ford

Cinvestav

Outline

1. Motivation
   - Mimicking the way Human Solve Problems
   - What is Search?
2. First Idea, State Space Problem
   - Introduction
   - Better Representation
     - Example
   - Solution Definition
   - Weighted State Space Problem
   - Evaluation of Search Strategies
   - Sparse Representation of Graphs
3. Uninformed Graph Search Algorithms
   - Implicit State Space Graph
   - Back to Implicit State Space Definition
   - Basic Functions
   - Depth-First Search
   - Breadth-First Search
   - Combining DFS and BFS
   - We Have the Results of Solving a Maze
4. Different ways of doing Stuff
   - What happened when you have weights?
   - What to do with negative weights?
   - Implicit Bellman-Ford

Cinvestav

Something is quite interesting to observe

Solving Problems as Humans

- It requires to start in some point and take an action to move to the next state.

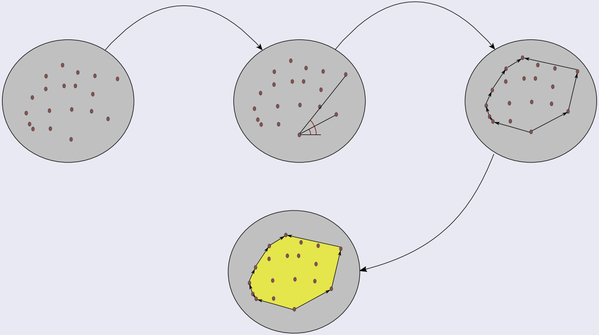
In mathematics, we do the following (Example Jarviz's Gift Wrapping Convex Hull)

Something is quite interesting to observe

Solving Problems as Humans

- It requires to start in some point and take an action to move to the next state.

In mathematics, we do the following (Example Jarviz's Gift Wrapping Convex Hull)



Therefore

Once one has established the initial policy (Cost Function) to solve the problem

- You can start designing a way to **search** for the possible solution.

Therefore

- The Concept of Search is the one that need to be explored in order to obtain an answer!!!

Therefore

Once one has established the initial policy (Cost Function) to solve the problem

- You can start designing a way to **search** for the possible solution.

Therefore

- The Concept of Search is the one that need to be explored in order to obtain an answer!!!

Outline

1. Motivation
   - Mimicking the way Human Solve Problems
   - What is Search?
2. First Idea, State Space Problem
   - Introduction
   - Better Representation
     - Example
   - Solution Definition
   - Weighted State Space Problem
   - Evaluation of Search Strategies
   - Sparse Representation of Graphs
3. Uninformed Graph Search Algorithms
   - Implicit State Space Graph
   - Back to Implicit State Space Definition
   - Basic Functions
   - Depth-First Search
   - Breadth-First Search
   - Combining DFS and BFS
   - We Have the Results of Solving a Maze
4. Different ways of doing Stuff
   - What happened when you have weights?
   - What to do with negative weights?
   - Implicit Bellman-Ford

Cinvestav

What is Search?

In computer Sciences

- **Every algorithm searches for the completion of a given task. [1]**

What is Search?

In computer Sciences

- **Every algorithm searches for the completion of a given task. [1]**

The process of problem solving can often be modeled as a search in a State Space.

1. A set of rules to move from a state to another state.
2. A state path that indicates our search in the State Space.
3. A Goal in such State Space.

- Looking for the best possible path.

What is Search?

In computer Sciences

- **Every algorithm searches for the completion of a given task. [1]**

The process of problem solving can often be modeled as a search in a State Space.

1. A set of rules to move from a state to another state.
2. A state path that indicates our search in the State Space.
3. A Goal in such State Space.

- Looking for the best possible path.

What is Search?

In computer Sciences

- **Every algorithm searches for the completion of a given task. [1]**

The process of problem solving can often be modeled as a search in a State Space.

1. A set of rules to move from a state to another state.
2. A state path that indicates our search in the State Space.
3. A Goal in such State Space.

- Looking for the best possible path.

Cinvestav

What is Search?

In computer Sciences

- **Every algorithm searches for the completion of a given task. [1]**

The process of problem solving can often be modeled as a search in a State Space.

1. A set of rules to move from a state to another state.
2. A state path that indicates our search in the State Space.
3. A Goal in such State Space.

- Looking for the best possible path.

Cinvestav

What is Search?

Example based in the idea of **Breadth First Search**

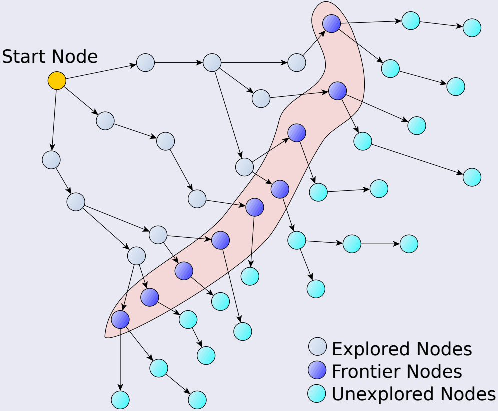


Figure: Example of Search

Outline

1. Motivation
   - Mimicking the way Human Solve Problems
   - What is Search?
2. First Idea, State Space Problem
   - Introduction
   - Better Representation
     - Example
   - Solution Definition
   - Weighted State Space Problem
   - Evaluation of Search Strategies
   - Sparse Representation of Graphs
3. Uninformed Graph Search Algorithms
   - Implicit State Space Graph
   - Back to Implicit State Space Definition
   - Basic Functions
   - Depth-First Search
   - Breadth-First Search
   - Combining DFS and BFS
   - We Have the Results of Solving a Maze
4. Different ways of doing Stuff
   - What happened when you have weights?
   - What to do with negative weights?
   - Implicit Bellman-Ford

Cinvestav

State Space Problem

State Space Problem [1]

Definition A state space problem $P = (S, A, s, T)$ consists of a:

1. Set of states S.
2. A starting state s
3. A set of goal states $T \subseteq S$.
4. A finite set of actions $A = \{a_1, a_2 ..., a_n\}$.
    1. Where $a_i : S \to S$ is a function that transform a state into another state.

State Space Problem

State Space Problem [1]

Definition A state space problem $P = (S, A, s, T)$ consists of a:

1. Set of states S.
2. A starting state s
3. A set of goal states $T \subseteq S$.
4. A finite set of actions $A = \{a_1, a_2 ..., a_n\}$.
   1. Where $a_i : S \rightarrow S$ is a function that transform a state into another state.

Cinvestav

State Space Problem

State Space Problem [1]

Definition A state space problem $P = (S, A, s, T)$ consists of a:

1. Set of states S.
2. A starting state s
3. A set of goal states $T \subseteq S$.
4. A finite set of actions $A = \{a_1, a_2 ..., a_n\}$.
   1. Where $a_i : S \rightarrow S$ is a function that transform a state into another state.

State Space Problem

State Space Problem [1]

Definition A state space problem $P = (S, A, s, T)$ consists of a:

1. Set of states S.
2. A starting state s
3. A set of goal states $T \subseteq S$.
4. A finite set of actions $A = \{a_1, a_2, \ldots, a_n\}$.
   1. Where $a_i : S \to S$ is a function that transform a state into another state.

State Space Problem

State Space Problem [1]

Definition A state space problem $P = (S, A, s, T)$ consists of a:

1. Set of states S.
2. A starting state s
3. A set of goal states $T \subseteq S$.
4. A finite set of actions $A = \{a_1, a_2..., a_n\}$.
   1. Where $a_i : S \to S$ is a function that transform a state into another state.

Cinvestav

State Space Problem

State Space Problem [1]

Definition A state space problem $P = (S, A, s, T)$ consists of a:

1. Set of states S.
2. A starting state s
3. A set of goal states $T \subseteq S$.
4. A finite set of actions $A = \{a_1, a_2 ..., a_n\}$.
    1. Where $a_i : S \to S$ is a function that transform a state into another state.

Example, Railroad Switching

Description

- An engine (**E**) at the siding can push or pull two cars (**A** and **B**) on the track.
- The railway passes through a tunnel that only the engine, but not the rail cars, can pass.

Goal

- To exchange the location of the two cars and have the engine back on the siding.

Cinvestav

Example, Railroad Switching

Description

- An engine (**E**) at the siding can push or pull two cars (**A** and **B**) on the track.
- The railway passes through a tunnel that only the engine, but not the rail cars, can pass.

Goal

- To exchange the location of the two cars and have the engine back on the siding.

Example, Railroad Switching

Description

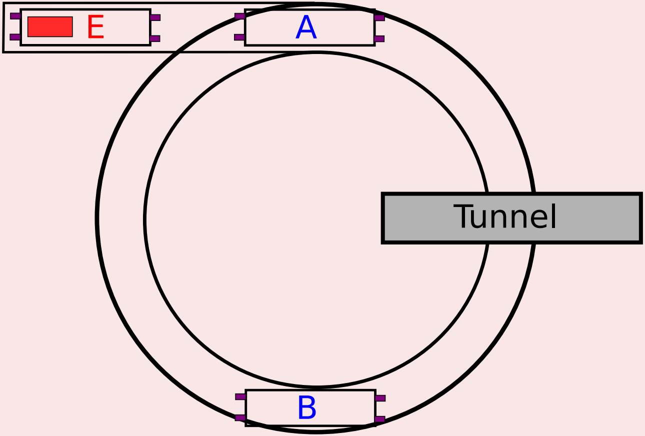
- An engine (**E**) at the siding can push or pull two cars (**A** and **B**) on the track.
- The railway passes through a tunnel that only the engine, but not the rail cars, can pass.

Goal

- To exchange the location of the two cars and have the engine back on the siding.

Example: RAILROAD SWITCHING

The Structue of the Problem



Outline

1. Motivation
   - Mimicking the way Human Solve Problems
   - What is Search?
2. First Idea, State Space Problem
   - Introduction
   - Better Representation
     - Example
   - Solution Definition
   - Weighted State Space Problem
   - Evaluation of Search Strategies
   - Sparse Representation of Graphs
3. Uninformed Graph Search Algorithms
   - Implicit State Space Graph
   - Back to Implicit State Space Definition
   - Basic Functions
   - Depth-First Search
   - Breadth-First Search
   - Combining DFS and BFS
   - We Have the Results of Solving a Maze
4. Different ways of doing Stuff
   - What happened when you have weights?
   - What to do with negative weights?
   - Implicit Bellman-Ford

Cinvestav

State Space Problem Graph

Definition

A problem graph $G = (V, E, s, T)$ for the state space problem $P = (S, A, s, T)$ is defined by:

1. $V = S$ as the set of nodes.
2. $s \in S$ as the initial node.
3. T as the set of goal nodes.
4. $E \subseteq V \times V$ as the set of edges that connect nodes to nodes with $(u, v) \in E$ if and only if there exists an $a \in A$ with $a(u) = v$.

State Space Problem Graph

Definition

A problem graph $G = (V, E, s, T)$ for the state space problem $P = (S, A, s, T)$ is defined by:

1. $V = S$ as the set of nodes.
2. $s \in S$ as the initial node.
3. T as the set of goal nodes.
4. $E \subseteq V \times V$ as the set of edges that connect nodes to nodes with $(u, v) \in E$ if and only if there exists an $a \in A$ with $a(u) = v$.

State Space Problem Graph

Definition

A problem graph $G = (V, E, s, T)$ for the state space problem $P = (S, A, s, T)$ is defined by:

1. $V = S$ as the set of nodes.
2. $s \in S$ as the initial node.
3. T as the set of goal nodes.
4. $E \subseteq V \times V$ as the set of edges that connect nodes to nodes with $(u, v) \in E$ if and only if there exists an $a \in A$ with $a(u) = v$.

State Space Problem Graph

Definition

A problem graph $G = (V, E, s, T)$ for the state space problem $P = (S, A, s, T)$ is defined by:

1. $V = S$ as the set of nodes.
2. $s \in S$ as the initial node.
3. T as the set of goal nodes.
4. $E \subseteq V \times V$ as the set of edges that connect nodes to nodes with $(u, v) \in E$ if and only if there exists an $a \in A$ with $a(u) = v$.

State Space Problem Graph

Definition

A problem graph $G = (V, E, s, T)$ for the state space problem $P = (S, A, s, T)$ is defined by:

1. $V = S$ as the set of nodes.
2. $s \in S$ as the initial node.
3. T as the set of goal nodes.
4. $E \subseteq V \times V$ as the set of edges that connect nodes to nodes with $(u, v) \in E$ if and only if there exists an $a \in A$ with $a(u) = v$.

Outline

1. Motivation
   - Mimicking the way Human Solve Problems
   - What is Search?
2. First Idea, State Space Problem
   - Introduction
   - Better Representation
     - Example
   - Solution Definition
   - Weighted State Space Problem
   - Evaluation of Search Strategies
   - Sparse Representation of Graphs
3. Uninformed Graph Search Algorithms
   - Implicit State Space Graph
   - Back to Implicit State Space Definition
   - Basic Functions
   - Depth-First Search
   - Breadth-First Search
   - Combining DFS and BFS
   - We Have the Results of Solving a Maze
4. Different ways of doing Stuff
   - What happened when you have weights?
   - What to do with negative weights?
   - Implicit Bellman-Ford

Cinvestav

Example

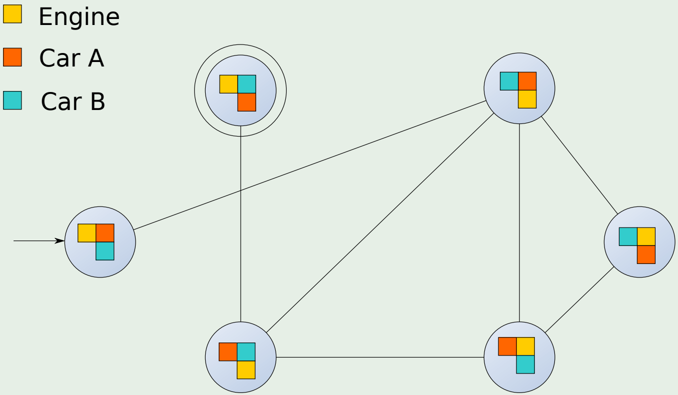
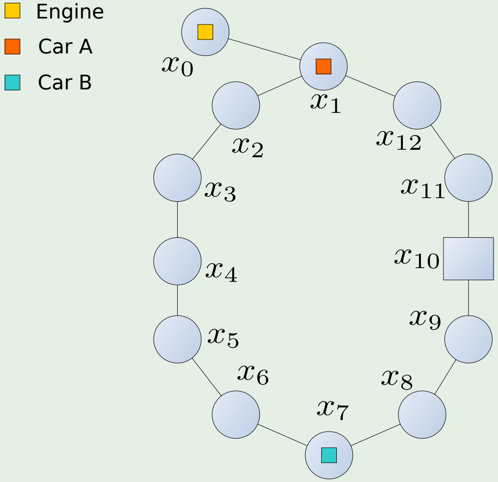


Figure: Possible states are labeled by the locations of the engine (E) and the cars (A and B), either in the form of a string or of a pictogram; EAB is the start state, EBA is the goal state.

Example

Inside of each state you could have



Outline

1. Motivation
   - Mimicking the way Human Solve Problems
   - What is Search?
2. First Idea, State Space Problem
   - Introduction
   - Better Representation
     - Example
   - Solution Definition
   - Weighted State Space Problem
   - Evaluation of Search Strategies
   - Sparse Representation of Graphs
3. Uninformed Graph Search Algorithms
   - Implicit State Space Graph
   - Back to Implicit State Space Definition
   - Basic Functions
   - Depth-First Search
   - Breadth-First Search
   - Combining DFS and BFS
   - We Have the Results of Solving a Maze
4. Different ways of doing Stuff
   - What happened when you have weights?
   - What to do with negative weights?
   - Implicit Bellman-Ford

Cinvestav

Solution

Definition

- A solution $\pi = (a_1, a_2, ..., a_k)$ is an ordered sequence of actions $a_i \in A, i \in 1, ..., k$ that transforms the initial state s into one of the goal states $t \in T$.

Thus

- There exists a sequence of states $u_i \in S, i \in 0, ..., k$, with $u_0 = s$, $u_k = t$, and u_i is the outcome of applying a_i to $u_{i-1}, i \in 1, ..., k$.

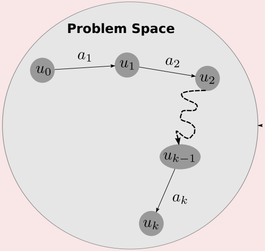
Solution

Definition

- A solution $\pi = (a_1, a_2, ..., a_k)$ is an ordered sequence of actions $a_i \in A, i \in 1, ..., k$ that transforms the initial state s into one of the goal states $t \in T$.

Thus

- There exists a sequence of states $u_i \in S, i \in 0, ..., k$, with $u_0 = s$, $u_k = t$, and u_i is the outcome of applying a_i to $u_{i-1}, i \in 1, ..., k$.



We want the following

We are interested in!!!

- **Solution length of a problem i.e.**
  - **the number of actions in the sequence.**
- Cost of the solution
  - Based on a Cost Function.

Cinvestav

We want the following

We are interested in!!!

- **Solution length of a problem i.e.**
  - **the number of actions in the sequence.**
- **Cost of the solution**
  - **Based on a Cost Function.**

Outline

1. Motivation
   - Mimicking the way Human Solve Problems
   - What is Search?
2. First Idea, State Space Problem
   - Introduction
   - Better Representation
     - Example
   - Solution Definition
   - Weighted State Space Problem
   - Evaluation of Search Strategies
   - Sparse Representation of Graphs
3. Uninformed Graph Search Algorithms
   - Implicit State Space Graph
   - Back to Implicit State Space Definition
   - Basic Functions
   - Depth-First Search
   - Breadth-First Search
   - Combining DFS and BFS
   - We Have the Results of Solving a Maze
4. Different ways of doing Stuff
   - What happened when you have weights?
   - What to do with negative weights?
   - Implicit Bellman-Ford

Cinvestav

It is more

As in Graph Theory

- We can add a weight to each edge

We can then

- Define the Weighted State Space Problem

It is more

As in Graph Theory

- We can add a weight to each edge

We can then

- Define the Weighted State Space Problem

Weighted State Space Problem

Definition

- A weighted state space problem is a tuple $P = (S, A, s, T, w)$, where w is a cost function$w : A \rightarrow \mathbb{R}$. The cost of a path consisting of actions $a_1, ..., a_n$ is defined as $\sum_{i=1}^{n} w\left(a_i\right)$.
- In a weighted search space, we call a **solution optimal**, if it has minimum cost among all feasible solutions.

Cinvestav

Weighted State Space Problem

Definition

- A weighted state space problem is a tuple $P = (S, A, s, T, w)$, where w is a cost function$w : A \to \mathbb{R}$. The cost of a path consisting of actions $a_1, ..., a_n$ is defined as $\sum_{i=1}^{n} w(a_i)$.
- In a weighted search space, we call a **solution optimal,** if it has minimum cost among all feasible solutions.

Then

Observations I

- For a weighted state space problem, there is a corresponding weighted problem graph $G = (V, E, s, T, w)$, where w is extended to $E \to \mathbb{R}$ in the straightforward way.
- The weight or cost of a path $\pi = (v_0, ..., v_k)$ is defined as $w(\pi) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.

Cinvestav

Then

Observations I

- For a weighted state space problem, there is a corresponding weighted problem graph $G = (V, E, s, T, w)$, where w is extended to $E \to \mathbb{R}$ in the straightforward way.
- The weight or cost of a path $\pi = (v_0, ..., v_k)$ is defined as $w(\pi) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.

Observations II

- $\delta(s, t) = \min \{w(\pi) | \pi = (v_0 = s, ..., v_k = t)\}$
- The optimal solution cost can be abbreviated as $\delta(s, T) = \min \{t \in T | \delta(s, t)\}$.

Cinvestav

Then

Observations I

- For a weighted state space problem, there is a corresponding weighted problem graph $G = (V, E, s, T, w)$, where w is extended to $E \to \mathbb{R}$ in the straightforward way.
- The weight or cost of a path $\pi = (v_0, ..., v_k)$ is defined as $w(\pi) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.

Observations II

- $\delta(s, t) = \min \{ w(\pi) | \pi = (v_0 = s, ..., v_k = t) \}$
- The optimal solution cost can be abbreviated as $\delta(s, T) = \min \{ t \in T | \delta(s, t) \}$.

Then

Observations I

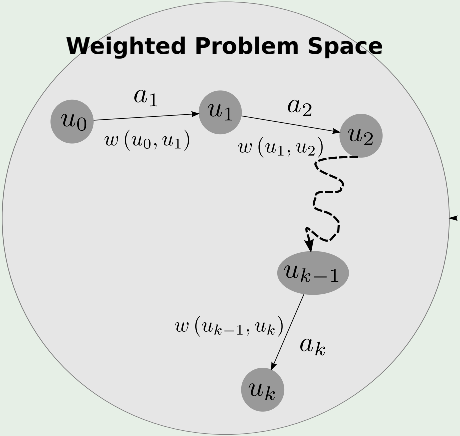
- For a weighted state space problem, there is a corresponding weighted problem graph $G = (V, E, s, T, w)$, where w is extended to $E \to \mathbb{R}$ in the straightforward way.
- The weight or cost of a path $\pi = (v_0, ..., v_k)$ is defined as $w(\pi) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.

Observations II

- $\delta(s, t) = \min \{w(\pi) | \pi = (v_0 = s, ..., v_k = t)\}$
- The optimal solution cost can be abbreviated as $\delta(s, T) = \min \{t \in T | \delta(s, t)\}$.

Example

The weights



Notes in Graph Representation

Terms

- Node expansion (a.k.a. node exploration):
  - Generation of all neighbors of a node u.
  - This nodes are called successors of u.
  - u is a parent or predecessor.

- u_n is a descendant of each node $u_0, \ldots, u_{n-1}$.
- Thus, ancestor and descendant refer to paths of possibly more than one edge.

Notes in Graph Representation

Terms

- Node expansion (a.k.a. node exploration):
  - Generation of all neighbors of a node u.
  - This nodes are called successors of u.
  - u is a parent or predecessor.

In addition...

- All nodes $u_0, ..., u_{n-1}$ are called antecessors of u.
- u is a descendant of each node $u_0, ..., u_{n-1}$.
- Thus, **ancestor** and **descendant** refer to paths of possibly more than one edge.

Cinvestav

Notes in Graph Representation

Terms

- Node expansion (a.k.a. node exploration):
  - Generation of all neighbors of a node u.
  - This nodes are called successors of u.
  - u is a parent or predecessor.

In addition...

- All nodes $u_0, ..., u_{n-1}$ are called antecessors of u.
- u is a descendant of each node $u_0, ..., u_{n-1}$.
- Thus, **ancestor** and **descendant** refer to paths of possibly more than one edge.

Notes in Graph Representation

Terms

- Node expansion (a.k.a. node exploration):
  - Generation of all neighbors of a node u.
  - This nodes are called successors of u.
  - u is a parent or predecessor.

In addition...

- All nodes $u_0, ..., u_{n-1}$ are called antecessors of u.
- u is a descendant of each node $u_0, ..., u_{n-1}$.
- Thus, **ancestor** and **descendant** refer to paths of possibly more than one edge.

Notes in Graph Representation

Terms

- Node expansion (a.k.a. node exploration):
  - Generation of all neighbors of a node u.
  - This nodes are called successors of u.
  - u is a parent or predecessor.

In addition...

- All nodes $u_0, ..., u_{n-1}$ are called antecessors of u.
- u is a descendant of each node $u_0, ..., u_{n-1}$.
- Thus, **ancestor** and **descendant** refer to paths of possibly more than one edge.

Cinvestav

Notes in Graph Representation

Terms

- Node expansion (a.k.a. node exploration):
  - Generation of all neighbors of a node u.
  - This nodes are called successors of u.
  - u is a parent or predecessor.

In addition...

- All nodes $u_0, ..., u_{n-1}$ are called antecessors of u.
- u is a descendant of each node $u_0, ..., u_{n-1}$.
- Thus, **ancestor** and **descendant** refer to paths of possibly more than one edge.

Cinvestav

Notes in Graph Representation

Terms

- Node expansion (a.k.a. node exploration):
  - Generation of all neighbors of a node u.
  - This nodes are called successors of u.
  - u is a parent or predecessor.

In addition...

- All nodes $u_0, ..., u_{n-1}$ are called antecessors of u.
- u is a descendant of each node $u_0, ..., u_{n-1}$.
- Thus, **ancestor** and **descendant** refer to paths of possibly more than one edge.

Cinvestav

Outline

1. Motivation
   - Mimicking the way Human Solve Problems
   - What is Search?
2. First Idea, State Space Problem
   - Introduction
   - Better Representation
     - Example
   - Solution Definition
   - Weighted State Space Problem
   - Evaluation of Search Strategies
   - Sparse Representation of Graphs
3. Uninformed Graph Search Algorithms
   - Implicit State Space Graph
   - Back to Implicit State Space Definition
   - Basic Functions
   - Depth-First Search
   - Breadth-First Search
   - Combining DFS and BFS
   - We Have the Results of Solving a Maze
4. Different ways of doing Stuff
   - What happened when you have weights?
   - What to do with negative weights?
   - Implicit Bellman-Ford

Cinvestav

Evaluation of Search Strategies

Completeness

- Does it always find a solution if one exists?

Time complexity

- How many nodes are generated?

Space complexity

- Maximum number of nodes in memory.

Evaluation of Search Strategies

Completeness

- Does it always find a solution if one exists?

Time complexity

- How many nodes are generated?

Space complexity

- Maximum number of nodes in memory.

Evaluation of Search Strategies

Completeness

- Does it always find a solution if one exists?

Time complexity

- How many nodes are generated?

Space complexity

- Maximum number of nodes in memory.

Evaluation of Search Strategies

Optimality

- Does it always find a least-cost solution?

Measuring Time and Space Complexity

Branching Factor

- b: Branching factor of a state is the number of successors it has.

If $Succ(u)$ abbreviates the successor set of a state $u \in S$

- Then the branching factor is $|Succ(u)|$
  - That is, cardinality of $Succ(u)$.

Depth of the Solution

- δ: Depth of the least-cost solution.
- m: Maximum depth of the state space (may be ∞).

Measuring Time and Space Complexity

Branching Factor

- b: Branching factor of a state is the number of successors it has.

If $Succ(u)$ abbreviates the successor set of a state $u \in S$

- Then the branching factor is $|Succ(u)|$
  - That is, cardinality of $Succ(u)$.

Depth of the Solution

- δ: Depth of the least-cost solution.
- m: Maximum depth of the state space (may be ∞).

Measuring Time and Space Complexity

Branching Factor

- b: Branching factor of a state is the number of successors it has.

If $Succ(u)$ abbreviates the successor set of a state $u \in S$

- Then the branching factor is $|Succ(u)|$
  - That is, cardinality of $Succ(u)$.

Depth of the Solution

- δ: Depth of the least-cost solution.
- m: Maximum depth of the state space (may be ∞).

Cinvestav

Outline

1. Motivation
   - Mimicking the way Human Solve Problems
   - What is Search?
2. First Idea, State Space Problem
   - Introduction
   - Better Representation
     - Example
   - Solution Definition
   - Weighted State Space Problem
   - Evaluation of Search Strategies
   - **Sparse Representation of Graphs**
3. Uninformed Graph Search Algorithms
   - Implicit State Space Graph
   - Back to Implicit State Space Definition
   - Basic Functions
   - Depth-First Search
   - Breadth-First Search
   - Combining DFS and BFS
   - We Have the Results of Solving a Maze
4. Different ways of doing Stuff
   - What happened when you have weights?
   - What to do with negative weights?
   - Implicit Bellman-Ford

There is a Duality

Between

- Graph representation as abstract collection of vertices and edges
- A sparse Adjacency Representation

Therefore, we can do the classic in Mathematics

- Use our Linear Algebra tools to solve Graphical Problems

However

- Matrices have not traditionally been used for practical computing with graphs,
  - Given that the 2D arrays are not efficient representation of them

There is a Duality

Between

- Graph representation as abstract collection of vertices and edges
- A sparse Adjacency Representation

Therefore, we can do the classic in Mathematics

- Use our Linear Algebra tools to solve Graphical Problems

However

- Matrices have not traditionally been used for practical computing with graphs,
    - Given that the 2D arrays are not efficient representation of them

There is a Duality

Between

- Graph representation as abstract collection of vertices and edges
- A sparse Adjacency Representation

Therefore, we can do the classic in Mathematics

- Use our Linear Algebra tools to solve Graphical Problems

However

- Matrices have not traditionally been used for practical computing with graphs,
  - Given that the 2D arrays are not efficient representation of them

However

New Data Structures are palliating such problems

- Then, a $G=(V,E)$ with N vertices and M edges, the $N \times N$ adjacency matrix A has the property:
  - $A(i,j)=1$, if $e_{ij} \in E$

Something Notable

- There is a duality between the matrix multiplication and breadth-first search

$$BFS(G,s) \Leftrightarrow A^T v, v(s)=1$$

However

New Data Structures are palliating such problems

- Then, a $G = (V, E)$ with N vertices and M edges, the $N \times N$ adjacency matrix A has the property:
  - $A(i, j) = 1$, if $e_{ij} \in E$

Something Notable

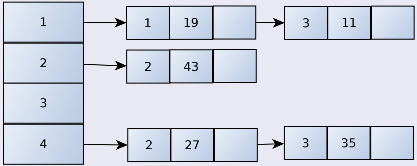
- There is a duality between the matrix multiplication and breadth-first search

$$BFS(G, s) \Leftrightarrow A^T \boldsymbol{v}, \boldsymbol{v}(s) = 1$$

Cinvestav

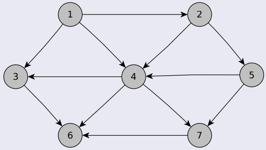
For this, we can use sparse structures

Adjacency Matrix



Here, we propose a new way of representing Graphs

Graphs can be represented by the use of Matrices



$$\begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix} \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Cinvestav

Why not to use Sparse Matrices?

We can have the following Coordinate Representation in lexicographic order

i	IA	JA	AA
1	1	2	1
2	1	3	1
3	1	4	1
4	2	4	1
5	2	5	1
6	3	6	1

i	IA	JA	AA
7	4	3	1
8	4	6	1
9	4	7	1
10	5	4	1
11	5	7	1
12	7	6	1

Cinvestav

Why not extend the data structure using linked list for iterators

Like

$1 \mid \longrightarrow \boxed{1} \longrightarrow \boxed{2} \longrightarrow \boxed{3}$

$2 \mid \longrightarrow \boxed{4} \longrightarrow \boxed{5}$

$4 \mid \longrightarrow \boxed{3} \longrightarrow \boxed{6} \longrightarrow \boxed{7}$

Cinvestav

Empty

Sparse_Matrix_bit_level(A, x)

1. $R = A$.iterRows() $\longrightarrow$ Use an iterator for the list of iterators
2. Z sparse vector
3. Do $I = R.next()$
4. $\quad Index = I.val$
5. $\quad Z[Index] = 0$
6. $\quad I = I.next()$
7. $\quad$ while $I! = Null$
8. $\quad\quad Z[Index] = Z[Index] + A.AA(I.val) * x(A.JA(I.val))$
9. $\quad\quad I = I.next()$
10. return Z

Cinvestav

Complexity

We have with K nonzero values in the matrix A

$$\sum_{it=1}^{m}\sum_{j_{it}}^{n} I\left(A\left(it, j_{it}\right) \neq 0\right) = O\left(K\right)$$

Then, we have the following

Matrix_BFS(A, s)

for $i = 1$ to V

  $distance\left[i\right] = 0$

$distance\left[s\right] = 1$

$front = distances$

for $i = 1$ to V

  $front =$Sparse_Matrix$(A, front)$ & $\neg distance$

  $nxt =$find$(front)$

  if $nxt = Null$

    break

  $distance\left(nxt\right) = i + 1$

$distance - = 1$

Here

$\neg distance$

$$\neg distance = \begin{cases} 1 & \text{if } distance\left[j\right] == 0 \\ 0 & \text{else} \end{cases}$$

Using Python notation

- find$(front)$ return the indexes that are not zero.

Here

$\neg distance$

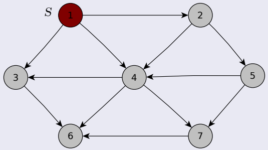
$$\neg distance = \begin{cases} 1 & \text{if } distance\left[j\right] == 0 \\ 0 & \text{else} \end{cases}$$

Using Python notation

- find$(front)$ return the indexes that are not zero.

We have

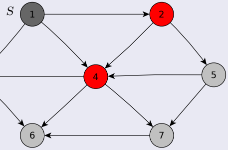
As you can see



$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}^T \begin{pmatrix} {\color{red}1} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Therefore, we have that

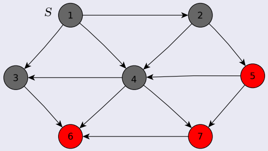
The following product



$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Now

The Next Step



$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Complexity

If we do not use rows on the graph, not used in the front expansion

- It is possible to reduce the complexity to

$$O(KV)$$

Making possible to have an efficient algorithms

- After all, we want efficiency.

Complexity

If we do not use rows on the graph, not used in the front expansion

- It is possible to reduce the complexity to

$$O\left(KV\right)$$

Making possible to have an efficient algorithms

- After all, we want efficiency.

Cinvestav

Outline

1. Motivation
   - Mimicking the way Human Solve Problems
   - What is Search?
2. First Idea, State Space Problem
   - Introduction
   - Better Representation
     - Example
   - Solution Definition
   - Weighted State Space Problem
   - Evaluation of Search Strategies
   - Sparse Representation of Graphs
3. Uninformed Graph Search Algorithms
   - Implicit State Space Graph
   - Back to Implicit State Space Definition
   - Basic Functions
   - Depth-First Search
   - Breadth-First Search
   - Combining DFS and BFS
   - We Have the Results of Solving a Maze
4. Different ways of doing Stuff
   - What happened when you have weights?
   - What to do with negative weights?
   - Implicit Bellman-Ford

Cinvestav

Implicit State Space Graph [1]

An Interesting Fact

- Solving state space problems is sometimes better characterized as a search in an implicit graph.

The difference is that not all edges have to be explicitly stored

- They are generated by a set of Rules.

This setting of an implicit generation of the search space

- It is also called **on-the-fly**, incremental, or **lazy state** space generation in some domains.

Cinvestav

Implicit State Space Graph [1]

An Interesting Fact

- Solving state space problems is sometimes better characterized as a search in an implicit graph.

The difference is that not all edges have to be explicitly stored

- They are generated by a set of Rules.

This setting of an implicit generation of the search space

- It is also called **on-the-fly**, incremental, or **lazy state** space generation in some domains.

Cinvestav

Implicit State Space Graph [1]

An Interesting Fact

- Solving state space problems is sometimes better characterized as a search in an implicit graph.

The difference is that not all edges have to be explicitly stored

- They are generated by a set of Rules.

This setting of an implicit generation of the search space

- It is also called **on-the-fly**, incremental, or **lazy state** space generation in some domains.

Here the following modification to the explicit Sparse Matrix

Add the necessary information (Nodes and Edges based on actions)

- A a new node is generated
  - You only need to update the possible edges

This allows to maintain a compact representation

- After all this was one of the main critiques that leaded to an AI Winder

Here the following modification to the explicit Sparse Matrix

Add the necessary information (Nodes and Edges based on actions)

- A a new node is generated
  - You only need to update the possible edges

This allows to maintain a compact representation

- After all this was one of the main critiques that leaded to an AI Winder

Outline

1. Motivation
   - Mimicking the way Human Solve Problems
   - What is Search?
2. First Idea, State Space Problem
   - Introduction
   - Better Representation
     - Example
   - Solution Definition
   - Weighted State Space Problem
   - Evaluation of Search Strategies
   - Sparse Representation of Graphs
3. Uninformed Graph Search Algorithms
   - Implicit State Space Graph
   - Back to Implicit State Space Definition
   - Basic Functions
   - Depth-First Search
   - Breadth-First Search
   - Combining DFS and BFS
   - We Have the Results of Solving a Maze
4. Different ways of doing Stuff
   - What happened when you have weights?
   - What to do with negative weights?
   - Implicit Bellman-Ford

Cinvestav

A More Complete Definition

Definition

In an implicit state space graph, we have

- An initial node $s \in V$.
- A set of goal nodes determined by a predicate

$$Goal : V \to B = \{false, true\}$$

- A node expansion procedure $Expand : V \to 2^V$.

A More Complete Definition

Definition

In an implicit state space graph, we have

- **An initial node** $s \in V$.
- **A set of goal nodes determined by a predicate**

$$Goal : V \rightarrow B = \{false, true\}$$

- **A node expansion procedure** $Expand : V \rightarrow 2^V$.

A More Complete Definition

Definition

In an implicit state space graph, we have

- **An initial node $s \in V$.**
- **A set of goal nodes determined by a predicate**

$$Goal : V \rightarrow B = \{false, true\}$$

- **A node expansion procedure $Expand : V \rightarrow 2^V$.**

A More Complete Definition

Definition

In an implicit state space graph, we have

- **An initial node $s \in V$.**
- **A set of goal nodes determined by a predicate**

$$Goal : V \rightarrow B = \{false, true\}$$

- **A node expansion procedure $Expand : V \rightarrow 2^{V}$.**

Open and Closed List

Reached Nodes

- They are divided into
  - Expanded Nodes - Closed List
  - Generated Nodes (Still not expanded) - Open List - Search Frontier

Cinvestav

Open and Closed List

Reached Nodes

- They are divided into
  - Expanded Nodes - Closed List
  - Generated Nodes (Still not expanded) - Open List - Search Frontier

Search Tree

The set of all explicitly generated paths rooted at the start node (leaves = Open Nodes) constitutes the search tree of the underlying problem graph.

Cinvestav

Open and Closed List

Reached Nodes

- They are divided into
  - Expanded Nodes - Closed List
  - Generated Nodes (Still not expanded) - Open List - Search Frontier

Search Tree

The set of all explicitly generated paths rooted at the start node (leaves = Open Nodes) constitutes the search tree of the underlying problem graph.

Cinvestav

Open and Closed List

Reached Nodes

- They are divided into
  - Expanded Nodes - Closed List
  - Generated Nodes (Still not expanded) - Open List - Search Frontier

Search Tree

The set of all explicitly generated paths rooted at the start node (leaves = Open Nodes) constitutes the search tree of the underlying problem graph.

Example

Problem Graph

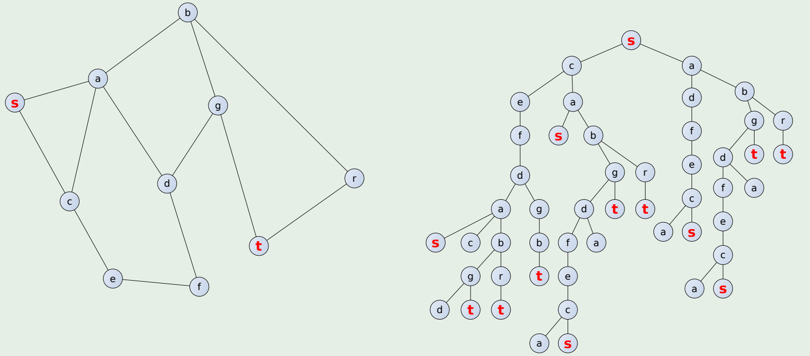


Figure: Problem Graph and Expansion Tree

Outline

1 Motivation
- Mimicking the way Human Solve Problems
- What is Search?

2 First Idea, State Space Problem
- Introduction
- Better Representation
  - Example
- Solution Definition
- Weighted State Space Problem
- Evaluation of Search Strategies
- Sparse Representation of Graphs

3 Uninformed Graph Search Algorithms
- Implicit State Space Graph
- Back to Implicit State Space Definition
- Basic Functions
- Depth-First Search
- Breadth-First Search
- Combining DFS and BFS
- We Have the Results of Solving a Maze

4 Different ways of doing Stuff
- What happened when you have weights?
- What to do with negative weights?
- Implicit Bellman-Ford

Cinvestav

Skeleton of a Search Algorithm

Basic Algorithm

Procedure Implicit-Graph-Search

Input: **Start node** s**, successor function** $Expand$ **and** $Goal$

Output: **Path from** s **to a goal node** $t \in T$ **or** $\emptyset$ **if no such path exist**

```
Closed = ∅
Open = {s}
while (Open ≠ ∅)
    Get u from Open
    Closed = Closed ∪ {u}
    if (Goal (u))
        return Path(u)
    Succ(u) =Expand(u)
    for each v ∈Succ(u)
        Improve(u, v)
return ∅
```

Skeleton of a Search Algorithm

Basic Algorithm

Procedure Implicit-Graph-Search

Input: **Start node s, successor function $Expand$ and $Goal$**

Output: **Path from s to a goal node $t \in T$ or $\emptyset$ if no such path exist**

```
Closed = ∅
Open = {s}
while (Open ≠ ∅)
    Get u from Open
    Closed = Closed ∪ {u}
    if (Goal(u))
        return Path(u)
    Succ(u) = Expand(u)
    for each v ∈ Succ(u)
        Improve(u, v)
return ∅
```

Skeleton of a Search Algorithm

Basic Algorithm

Procedure Implicit-Graph-Search

Input: **Start node s, successor function $Expand$ and $Goal$**

Output: **Path from s to a goal node $t \in T$ or $\emptyset$ if no such path exist**

$Closed = \emptyset$
$Open = \{s\}$
while $(Open \neq \emptyset)$
    Get u **from** $Open$
    $Closed = Closed \cup \{u\}$
    if $(Goal\,(u))$
        return Path(u)
    Succ(u) =**Expand**(u)
    for each $v \in$**Succ**(u)
        Improve(u, v)
return $\emptyset$

Skeleton of a Search Algorithm

Basic Algorithm

Procedure Implicit-Graph-Search

Input: **Start node** s**, successor function** $Expand$ **and** $Goal$

Output: **Path from** s **to a goal node** $t \in T$ **or** $\emptyset$ **if no such path exist**

1. $Closed = \emptyset$
2. $Open = \{s\}$
3. **while** $(Open \neq \emptyset)$
4. **Get** u **from** $Open$
5. $Closed = Closed \cup \{u\}$
6. **if** $(Goal\,(u))$
7. **return Path**(u)
8. **Succ**(u) =**Expand**(u)
9. **for each** $v \in$**Succ**(u)
10. **Improve**(u, v)
11. **return** $\emptyset$

Skeleton of a Search Algorithm

Basic Algorithm

Procedure Implicit-Graph-Search

Input: **Start node** s**, successor function** $Expand$ **and** $Goal$

Output: **Path from** s **to a goal node** $t \in T$ **or** $\emptyset$ **if no such path exist**

1. $Closed = \emptyset$
2. $Open = \{s\}$
3. **while** $(Open \neq \emptyset)$
4. **Get** u **from** $Open$
5. $Closed = Closed \cup \{u\}$
6. **if** $(Goal\ (u))$
7. **return Path**(u)
8. **Succ**(u) =**Expand**(u)
9. **for each** $v \in$**Succ**(u)
10. **Improve**(u, v)
11. **return** $\emptyset$

Skeleton of a Search Algorithm

Basic Algorithm

Procedure Implicit-Graph-Search

Input: **Start node s, successor function $Expand$ and $Goal$**

Output: **Path from s to a goal node $t \in T$ or $\emptyset$ if no such path exist**

1. $Closed = \emptyset$
2. $Open = \{s\}$
3. **while** $(Open \neq \emptyset)$
4. **Get** u **from** $Open$
5. $Closed = Closed \cup \{u\}$
6. **if** $(Goal\ (u))$
7. **return Path**(u)
8. **Succ**(u) =**Expand**(u)
9. **for each** $v \in$**Succ**(u)
10. **Improve**(u, v)
11. **return** $\emptyset$

Skeleton of a Search Algorithm

Basic Algorithm

Procedure Implicit-Graph-Search

Input: **Start node** s**, successor function** $Expand$ **and** $Goal$

Output: **Path from** s **to a goal node** $t \in T$ **or** $\emptyset$ **if no such path exist**

1. $Closed = \emptyset$
2. $Open = \{s\}$
3. **while** $(Open \neq \emptyset)$
4.   **Get** u **from** $Open$
5. $Closed = Closed \cup \{u\}$
6.   **if** $(Goal\,(u))$
7.   **return Path**(u)
8.   **Succ**(u) =**Expand**(u)
9.   **for each** $v \in$**Succ**(u)
10.   **Improve**(u, v)
11. **return** $\emptyset$

Skeleton of a Search Algorithm

Basic Algorithm

Procedure Implicit-Graph-Search

Input: **Start node** s**, successor function** $Expand$ **and** $Goal$

Output: **Path from** s **to a goal node** $t \in T$ **or** $\emptyset$ **if no such path exist**

1. $Closed = \emptyset$
2. $Open = \{s\}$
3. **while** $(Open \neq \emptyset)$
4.   **Get** u **from** $Open$
5. $Closed = Closed \cup \{u\}$
6.   **if** $(Goal\,(u))$
7.   **return Path**(u)
8.   **Succ**(u) $=$**Expand**(u)
9.   **for each** $v \in$**Succ**(u)
10.   **Improve**(u, v)
11. **return** $\emptyset$

Skeleton of a Search Algorithm

Basic Algorithm

Procedure Implicit-Graph-Search

Input: **Start node** s**, successor function** $Expand$ **and** $Goal$

Output: **Path from** s **to a goal node** $t \in T$ **or** $\emptyset$ **if no such path exist**

```
Closed = ∅
Open = {s}
while (Open ≠ ∅)
    Get u from Open
    Closed = Closed ∪ {u}
    if (Goal(u))
        return Path(u)
    Succ(u) = Expand(u)
    for each v ∈ Succ(u)
        Improve(u, v)
return ∅
```

Skeleton of a Search Algorithm

Basic Algorithm

Procedure Implicit-Graph-Search

Input: **Start node** s**, successor function** $Expand$ **and** $Goal$

Output: **Path from** s **to a goal node** $t \in T$ **or** $\emptyset$ **if no such path exist**

```
Closed = ∅
Open = {s}
while (Open ≠ ∅)
    Get u from Open
    Closed = Closed ∪ {u}
    if (Goal(u))
        return Path(u)
    Succ(u) = Expand(u)
    for each v ∈ Succ(u)
        Improve(u, v)
return ∅
```

Improve Algorithm

Basic Algorithm

Improve

Input: **Nodes** u **and** v, v **successor of** u

Output: **Update parent** v, $Open$ **and** $Closed$

1. **if** $(v \notin Closed \cup Open)$
2. **Insert** v **into** $Open$
3. $parent\,(v) = u$

Cinvestav

Returning the Path

Basic Algorithm

Procedure Path

Input: **Node u, start node s and parents set by the algorithm**

Output: **Path from s to u**

1. $Path = Path \cup \{u\}$
2. **while** $(parent\,(u) \neq s)$
3. $\quad u = parent\,(u)$
4. $\quad Path = Path \cup \{u\}$

Cinvestav

Algorithms to be Explored

Algorithm

1. Depth-First Search
2. Breadth-First Search
3. Dijkstra's Algorithm
4. Relaxed Node Selection
5. Bellman-Ford
6. Dynamic Programming

Outline

1 Motivation
- Mimicking the way Human Solve Problems
- What is Search?

2 First Idea, State Space Problem
- Introduction
- Better Representation
  - Example
- Solution Definition
- Weighted State Space Problem
- Evaluation of Search Strategies
- Sparse Representation of Graphs

3 Uninformed Graph Search Algorithms
- Implicit State Space Graph
- Back to Implicit State Space Definition
- Basic Functions
- Depth-First Search
- Breadth-First Search
- Combining DFS and BFS
- We Have the Results of Solving a Maze

4 Different ways of doing Stuff
- What happened when you have weights?
- What to do with negative weights?
- Implicit Bellman-Ford

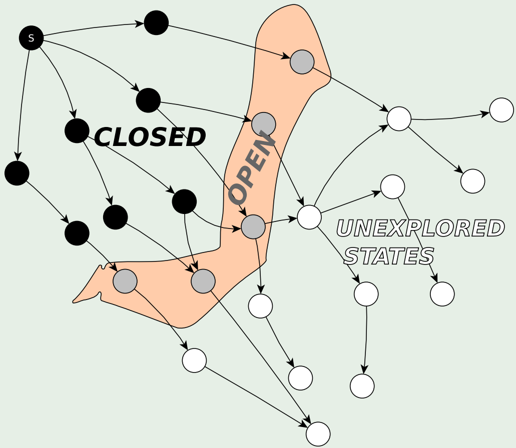
Depth First Search (DFS) [2]

Implementation

- Open List uses a Stack
  - Insert == Push
  - Select == Pop
  - Open == Stack
  - Closed == Set

Example of the Implicit Graph

Something Notable



By The Way

Did you notice the following? Given X a search space

- Open $\cap$ Closed $==\emptyset$
- $X-$(Open $\cup$ Closed) $\cap$ Open $==\emptyset$
- $X-$(Open $\cup$ Closed) $\cap$ Closed $==\emptyset$

By The Way

Did you notice the following? Given X a search space

- Open $\cap$ Closed $== \emptyset$
- $X -$(Open $\cup$ Closed) $\cap$ Open $== \emptyset$
- $X -$(Open $\cup$ Closed) $\cap$ Closed $== \emptyset$

Disjoint Set Representation

- Yes!!! We can do it!!!
- For the $Closed$ set!!!

Cinvestav

How DFS measures?

Complete?

- **No: fails in infinite-depth spaces or spaces with loops (If you allow node repetition)!!!**

Cinvestav

How DFS measures?

Complete?

- **No: fails in infinite-depth spaces or spaces with loops (If you allow node repetition)!!!**

Modify to avoid repeated states along path

- **However, you still have a problem What if you only store the search frontier?**
- **Ups!!! We have a problem... How do we recognize repeated states in complex search spaces?**

Cinvestav

How DFS measures?

Complete?

- No: fails in infinite-depth spaces or spaces with loops (If you allow node repetition)!!!

Modify to avoid repeated states along path

- However, you still have a problem What if you only store the search frontier?
- Ups!!! We have a problem... How do we recognize repeated states in complex search spaces?

Nevertheless

- Complete in finite spaces

How DFS measures?

Complete?

- No: fails in infinite-depth spaces or spaces with loops (If you allow node repetition)!!!

Modify to avoid repeated states along path

- However, you still have a problem What if you only store the search frontier?
- Ups!!! We have a problem... How do we recognize repeated states in complex search spaces?

Nevertheless

- Complete in finite spaces

Time?

It depends a lot on the representation an data structure representation

- In the case of adjacency lists for graph representation.

If we do not have repetitions

- $O(V+E) = O(E)$ and $|V| \ll |E|$

Given the branching b

- $O(b^m)$: terrible if m is much larger than δ, but if solutions are dense, may be much faster than breadth-first search

Time?

It depends a lot on the representation an data structure representation

- In the case of adjacency lists for graph representation.

If we do not have repetitions

- $O(V+E)=O(E)$ **and** $|V| \ll |E|$

Given the branching b

- $O(b^m)$: terrible if m is much larger than δ, but if solutions are dense, may be much faster than breadth-first search

Time?

It depends a lot on the representation an data structure representation

- In the case of adjacency lists for graph representation.

If we do not have repetitions

- $O(V+E) = O(E)$ **and** $|V| \ll |E|$

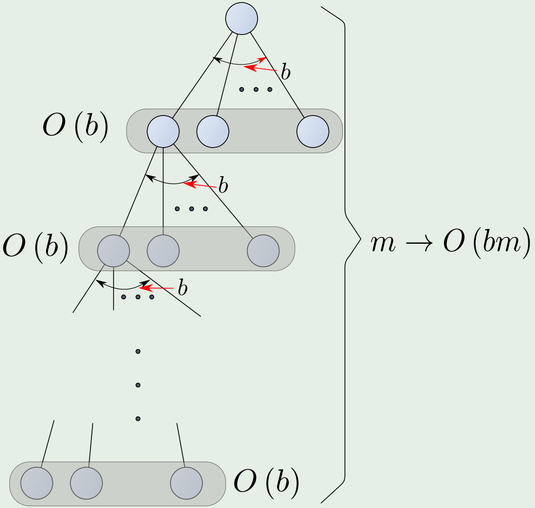
Given the branching b

- $O(b^m)$: **terrible if m is much larger than δ, but if solutions are dense, may be much faster than breadth-first search**

Cinvestav

What about the Space Complexity and Optimality?

Maintaining only the frontier



Optimal? No, look at the following example...

Example

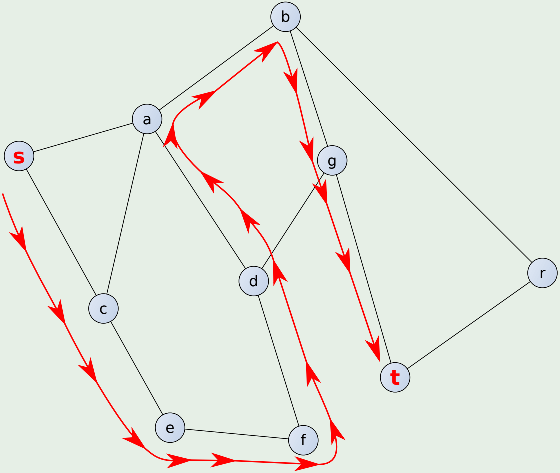


Figure: Goal at t from source node s

The Pseudo-Code - Solving the Problem of Repeated Nodes

Code - Iterative Version - Solving the Repetition of Nodes

DFS-Iterative(s)

Input: **start node** s**, set of** $Goals$

1. **Given** s **an starting node**
2. $Open$ **is a stack**
3. $Closed$ **is a set**
4. **Open.Push**(s)
5. $Closed = \emptyset$
6. **while** $Open \neq \emptyset$
7. v**=Open.pop()**
8. **if** $Closed \neq Closed \cup (v)$
9. **if** $v \in Goal$ **return** $Path(v)$
10. $succ(v) = Expand(v)$
11. **for each vertex** $u \in succ(v)$
12. **if** $Closed \neq Closed \cup (u)$
13. **Open.push**(u)

The Pseudo-Code - Solving the Problem of Repeated Nodes

Code - Iterative Version - Solving the Repetition of Nodes

DFS-Iterative(s)

Input: **start node** s**, set of** $Goals$

1. **Given** s **an starting node**
2. $Open$ **is a stack**
3. $Closed$ **is a set**
4. **Open.Push(**s**)**
5. $Closed = \emptyset$
6. **while** $Open \neq \emptyset$
7. v**=Open.pop()**
8. **if** $Closed \neq Closed \cup (v)$
9. **if** $v \in Goal$ **return** $Path(v)$
10. $succ(v) = Expand(v)$
11. **for each vertex** $u \in succ(v)$
12. **if** $Closed \neq Closed \cup (u)$
13. **Open.push(**u**)**

The Pseudo-Code - Solving the Problem of Repeated Nodes

Code - Iterative Version - Solving the Repetition of Nodes

DFS-Iterative(s)

Input: **start node** s**, set of** $Goals$

1. **Given** s **an starting node**
2. $Open$ **is a stack**
3. $Closed$ **is a set**
4. **Open.Push(**s**)**
5. $Closed = \emptyset$
6. **while** $Open \neq \emptyset$
7. $\quad$ v**=Open.pop()**
8. $\quad$ if $Closed \neq Closed \cup (v)$
9. $\quad\quad$ if $v \in Goal$ return $Path(v)$
10. $\quad\quad$ $succ(v) = Expand(v)$
11. $\quad\quad$ for each vertex $u \in succ(v)$
12. $\quad\quad\quad$ if $Closed \neq Closed \cup (u)$
13. $\quad\quad\quad\quad$ Open.push(u)

The Pseudo-Code - Solving the Problem of Repeated Nodes

Code - Iterative Version - Solving the Repetition of Nodes

DFS-Iterative(s)

Input: **start node** s**, set of** $Goals$

1. **Given** s **an starting node**
2. $Open$ **is a stack**
3. $Closed$ **is a set**
4. **Open.Push(**s**)**
5. $Closed = \emptyset$
6. **while** $Open \neq \emptyset$
7. v**=Open.pop()**
8.   **if** $Closed \neq Closed \cup (v)$
9.   **if** $v \in Goal$ **return** $Path(v)$
10. $succ(v) = Expand(v)$
11.   **for each vertex** $u \in succ(v)$
12.   **if** $Closed \neq Closed \cup (u)$
13.   **Open.push(**u**)**

The Pseudo-Code - Solving the Problem of Repeated Nodes

Code - Iterative Version - Solving the Repetition of Nodes

DFS-Iterative(s)

Input: **start node** s**, set of** $Goals$

1. **Given** s **an starting node**
2. $Open$ **is a stack**
3. $Closed$ **is a set**
4. **Open.Push(**s**)**
5. $Closed = \emptyset$
6. **while** $Open \neq \emptyset$
7. $\quad v$**=Open.pop()**
8. $\quad$ **if** $Closed \neq Closed \cup (v)$
9. $\quad\quad$ **if** $v \in Goal$ **return** $Path(v)$
10. $\quad\quad succ(v) = Expand(v)$
11. $\quad\quad$ **for each vertex** $u \in succ(v)$
12. $\quad\quad\quad$ **if** $Closed \neq Closed \cup (u)$
13. $\quad\quad\quad\quad$ **Open.push(**u**)**

The Pseudo-Code - Solving the Problem of Repeated Nodes

Code - Iterative Version - Solving the Repetition of Nodes

DFS-Iterative(s)

Input: **start node** s**, set of** $Goals$

- **Given** s **an starting node**
- $Open$ **is a stack**
- $Closed$ **is a set**
- **Open.Push(**s**)**
- $Closed = \emptyset$
- **while** $Open \neq \emptyset$
  - v**=Open.pop()**
  - **if** $Closed \neq Closed \cup (v)$
    - **if** $v \in Goal$ **return** $Path(v)$
    - $succ(v) = Expand(v)$
    - **for each vertex** $u \in succ(v)$
      - **if** $Closed \neq Closed \cup (u)$
        - **Open.push(**u**)**

Disjoint Set Representation

Using our Disjoint Set Representation

We get the ability to be able to compare two sets through the representatives!!!

Not only that

Using that, we solve the problem of node repetition

Little Problem

If we are only storing the frontier our disjoint set representation is not enough!!!

- More research is needed!!!

Disjoint Set Representation

Using our Disjoint Set Representation

We get the ability to be able to compare two sets through the representatives!!!

Not only that

Using that, we solve the problem of node repetition

Little Problem

If we are only storing the frontier our disjoint set representation is not enough!!!

- More research is needed!!!

Disjoint Set Representation

Using our Disjoint Set Representation

We get the ability to be able to compare two sets through the representatives!!!

Not only that

Using that, we solve the problem of node repetition

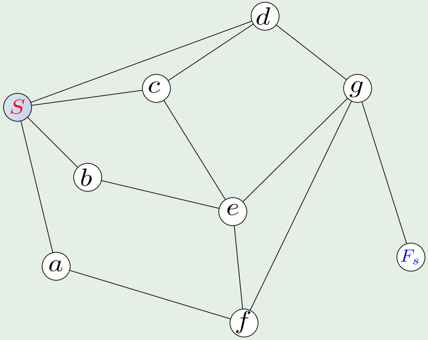
Little Problem

If we are only storing the frontier our disjoint set representation is not enough!!!

- More research is needed!!!

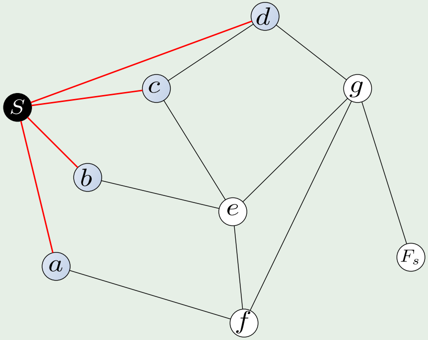
Example

Example



Example

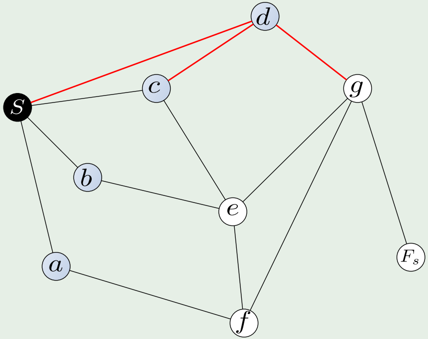
Example



Step	Selection	Open	Closed	Remarks
1	$\{\}$	$\{S\}$	$\{\}$	Push start node into the Stack
2	S	$\{d, c, b, a\}$	$\{S\}$	

Example

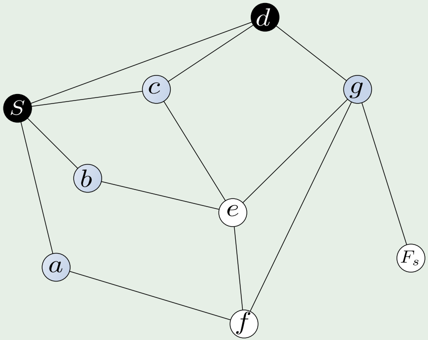
Example



Step	Selection	Open	Closed	Remarks
3	$\{d\}$	$\{g, c, b, a\}$	$\{S\}$	S and c are repeated
4	$\{g\}$	$c, b, a]$	$\{S, d\}$	

Example

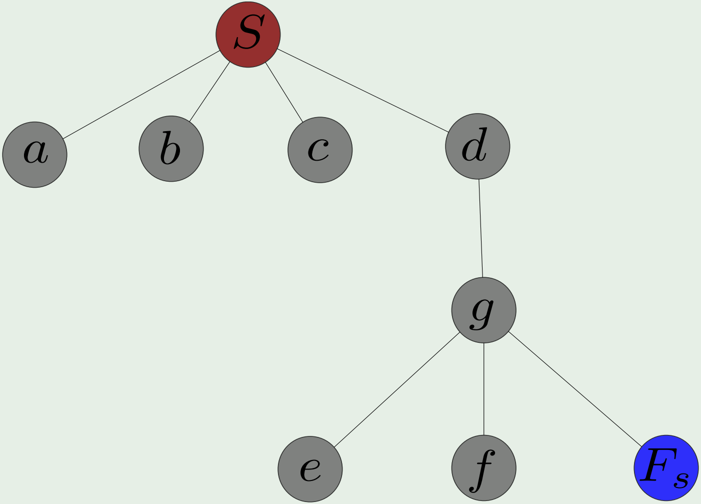
Example



Step	Selection	Open	Closed	Remarks
4	$\{g\}$	$\{c, b, a\}$	$\{S, d\}$	

The Depth-First Search Tree

With the following tree expantion



Outline

1. Motivation
   - Mimicking the way Human Solve Problems
   - What is Search?
2. First Idea, State Space Problem
   - Introduction
   - Better Representation
     - Example
   - Solution Definition
   - Weighted State Space Problem
   - Evaluation of Search Strategies
   - Sparse Representation of Graphs
3. Uninformed Graph Search Algorithms
   - Implicit State Space Graph
   - Back to Implicit State Space Definition
   - Basic Functions
   - Depth-First Search
   - Breadth-First Search
   - Combining DFS and BFS
   - We Have the Results of Solving a Maze
4. Different ways of doing Stuff
   - What happened when you have weights?
   - What to do with negative weights?
   - Implicit Bellman-Ford

Cinvestav

Bradth-First Search (BFS) [2]

Implementation by Adjacency List

- **Open List uses a Queue**
  - Insert == Enqueue
  - Select == Dequeue
  - Open == Queue
  - Closed == Set

Breast-First Search Pseudo-Code

BFS-Implicit(s)

Input: start node s, set of $Goals$

$Open$ **is a queue**
$Closed$ **is a set**
$Open$**.enqueue(**s**)**
$Closed = \emptyset$
while $Open \neq \emptyset$
    $v = Open$**.dequeue()**
    if $Closed \neq Closed \cup (v)$
        if $v \in Goal$ **return** $Path(v)$
        $succ(v) = Expand(v)$
        for each vertex $u \in succ(v)$
            if $Closed \neq Closed \cup (u)$
                Open.enqueue(u**)**

How BFS measures?

Evaluation

- **Complete? Yes if b is finite**
- **Time?** $1 + b + b^2 + b^3 + \ldots + b^\delta = O(b^\delta)$
- **Space?** $O\left(b^\delta\right)$ **This is a big problem**
- **Optimal? Yes, If cost is equal for each step.**

Cinvestav

How BFS measures?

Evaluation

- **Complete? Yes if b is finite**
- **Time?** $1 + b + b^2 + b^3 + \ldots + b^\delta = O(b^\delta)$
- **Space?** $O\left(b^\delta\right)$ **This is a big problem**
- **Optimal? Yes, If cost is equal for each step.**

Cinvestav

How BFS measures?

Evaluation

- **Complete? Yes if b is finite**
- **Time?** $1 + b + b^2 + b^3 + \ldots + b^\delta = O(b^\delta)$
- **Space?** $O\left(b^\delta\right)$ **This is a big problem**
- Optimal? Yes, If cost is equal for each step.

How BFS measures?

Evaluation

- **Complete? Yes if b is finite**
- **Time?** $1 + b + b^2 + b^3 + \ldots + b^\delta = O(b^\delta)$
- **Space?** $O\left(b^\delta\right)$ **This is a big problem**
- **Optimal? Yes, If cost is equal for each step.**

Question

Can we re-implement this in a different way?

- Linear Algebra Style?

What about such Complexity?

- Can we calculate such thing?

Question

Can we re-implement this in a different way?

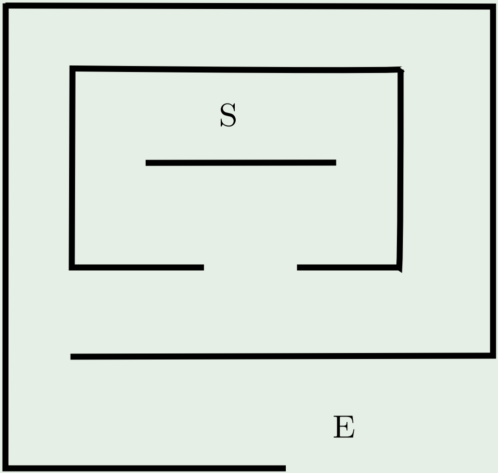
- Linear Algebra Style?

What about such Complexity?

- Can we calculate such thing?

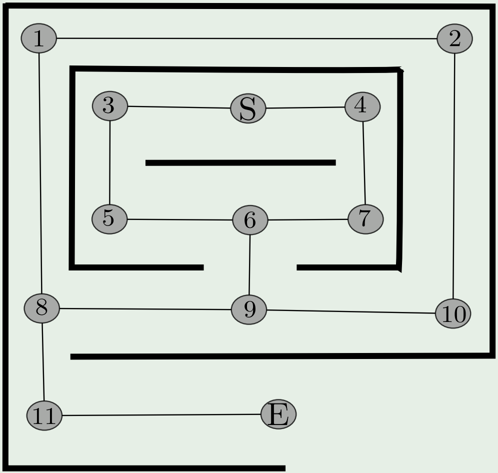
Example

Example



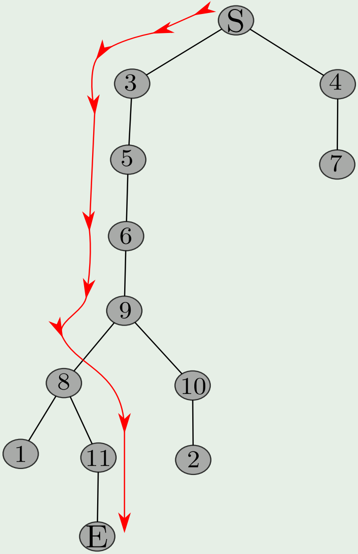
Example

Over-impose a Graph and take a look at the board



Example

With Breadth First Search Tree



Outline

1. Motivation
   - Mimicking the way Human Solve Problems
   - What is Search?
2. First Idea, State Space Problem
   - Introduction
   - Better Representation
     - Example
   - Solution Definition
   - Weighted State Space Problem
   - Evaluation of Search Strategies
   - Sparse Representation of Graphs
3. Uninformed Graph Search Algorithms
   - Implicit State Space Graph
   - Back to Implicit State Space Definition
   - Basic Functions
   - Depth-First Search
   - Breadth-First Search
   - Combining DFS and BFS
   - We Have the Results of Solving a Maze
4. Different ways of doing Stuff
   - What happened when you have weights?
   - What to do with negative weights?
   - Implicit Bellman-Ford

Cinvestav

Can we combine the benefits of both algorithms?

First Limit the Depth

- Depth-Limited Search (DLS) is an uninformed search.
- It is DFS imposing a maximum limit on the depth of the search.

return *node*

for each *child* in expand(*node*)

DLS(*child*, *goal*, *depth* − 1)

Can we combine the benefits of both algorithms?

First Limit the Depth

- Depth-Limited Search (DLS) is an uninformed search.
- It is DFS imposing a maximum limit on the depth of the search.

Algorithm

DLS$(node, goal, depth)$

1. if ($depth \geq 0$)
2. if ($node == goal$)
3. return $node$
4. for each $child$ in expand($node$)
5. DLS$(child, goal, depth - 1)$

Can we combine the benefits of both algorithms?

First Limit the Depth

- Depth-Limited Search (DLS) is an uninformed search.
- It is DFS imposing a maximum limit on the depth of the search.

Algorithm

DLS($node, goal, depth$)

1. if ($depth \geq 0$)
2. if ($node == goal$)
3. return $node$
4. for each $child$ in expand($node$)
5. DLS($child, goal, depth - 1$)

IMPORTANT!!!

- If $depth < \delta$ we will never find the answer!!!

Can we combine the benefits of both algorithms?

First Limit the Depth

- Depth-Limited Search (DLS) is an uninformed search.
- It is DFS imposing a maximum limit on the depth of the search.

Algorithm

DLS($node, goal, depth$)

1. if ($depth \geq 0$)
2.   if ($node == goal$)
3.   return $node$
4.   for each $child$ in expand($node$)
5.   DLS($child, goal, depth - 1$)

IMPORTANT!!!

- If $depth < \delta$ we will never find the answer!!!

Can we combine the benefits of both algorithms?

First Limit the Depth

- Depth-Limited Search (DLS) is an uninformed search.
- It is DFS imposing a maximum limit on the depth of the search.

Algorithm

DLS($node, goal, depth$)

1. if ($depth \geq 0$)
2. if ($node == goal$)
3. return $node$
4. for each $child$ in expand($node$)
5. DLS($child, goal, depth - 1$)

IMPORTANT!!!

- If $depth < \delta$ we will never find the answer!!!

We can do much more!!!

Iterative Deepening Search (IDS) [3]

- We can increment the depth in each run until we find the

Algorithm

IDS($node, goal$)

1. for $D = 0$ to ∞ : Step Size L
2. $\quad result = \text{DLS}(node, goal, D)$
3. $\quad$ if $result == goal$
4. $\quad\quad$ return $result$

We can do much more!!!

Iterative Deepening Search (IDS) [3]

- We can increment the depth in each run until we find the

Algorithm

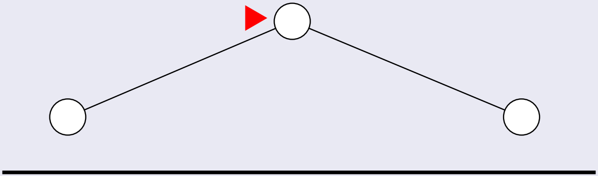
IDS$(node, goal)$

1. for $D = 0$ to ∞ : Step Size L
2. $\quad result = \text{DLS}(node, goal, D)$
3. $\quad$ if $result == goal$
4. $\quad\quad$ return $result$

Cinvestav

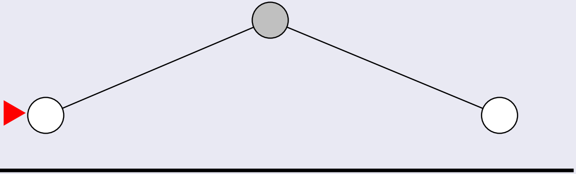
Example

Example: $D == 1$



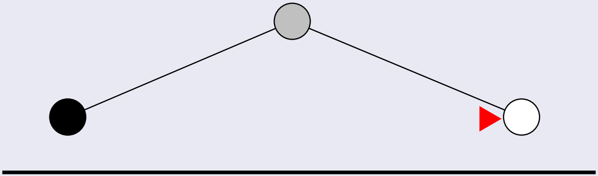
Example

Example: $D == 1$



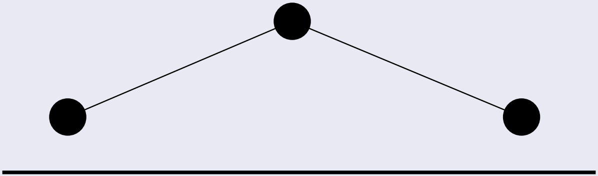
Example

Example: $D == 1$



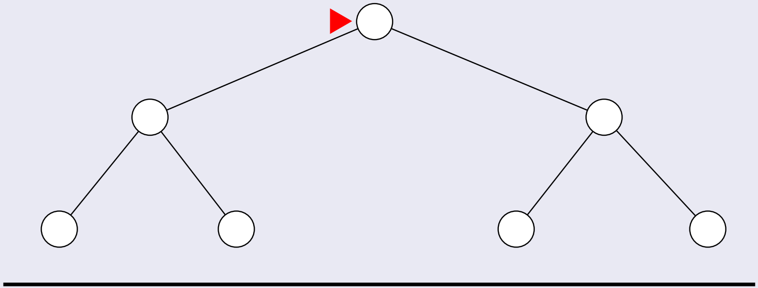
Example

Example: $D == 1$



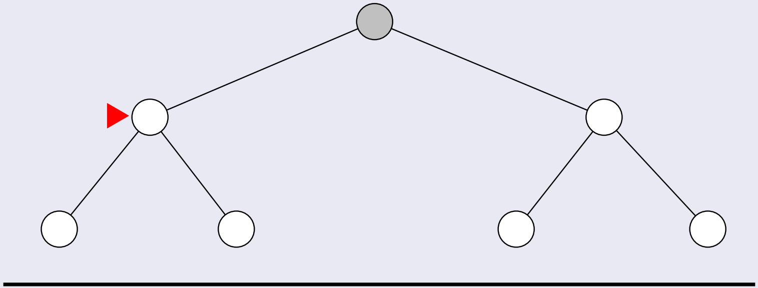
Example

Example: $D == 2$



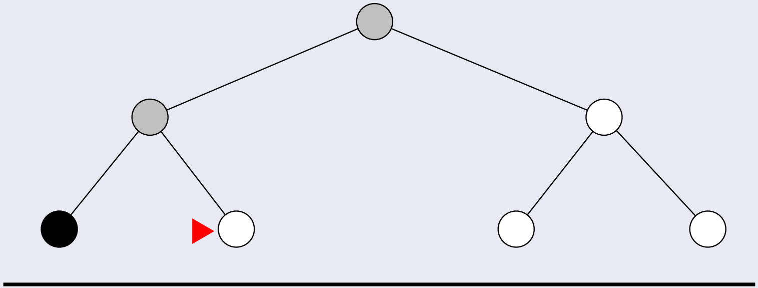
Example

Example: $D == 2$



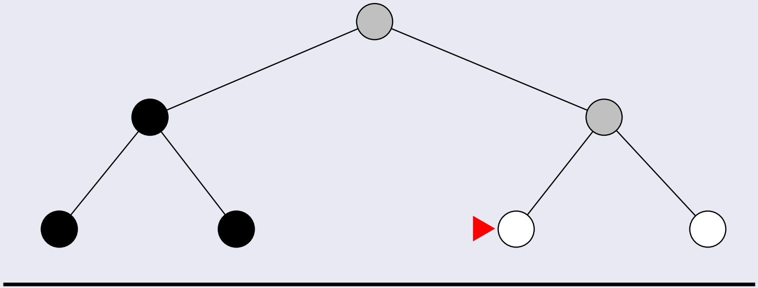
Example

Example: $D == 2$



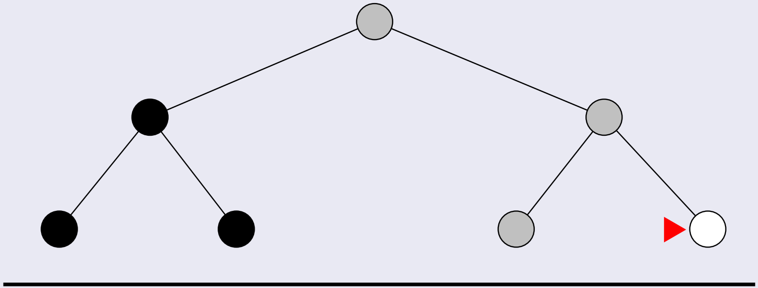
Example

Example: $D == 2$



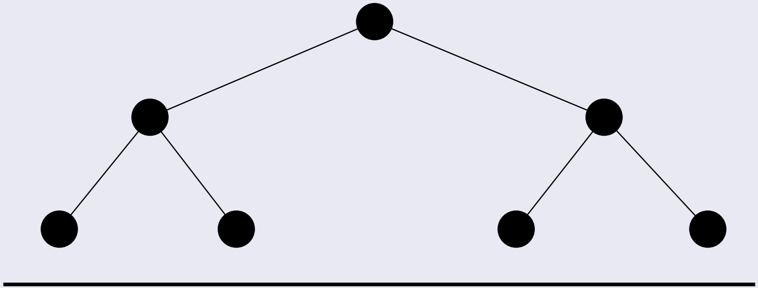
Example

Example: $D == 2$



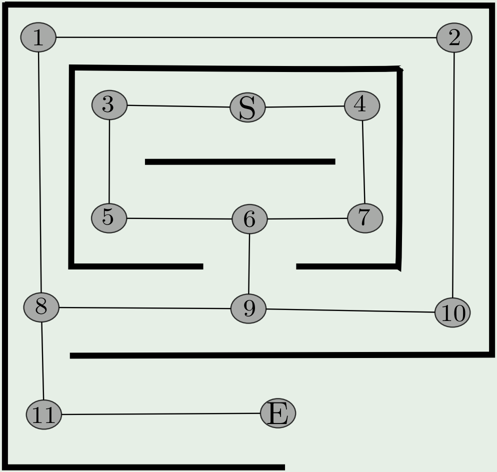
Example

Example: $D == 2$



Example

Over-impose a Graph and take a look at the board



Properties of IDS

Properties

- **Complete?** Yes
- **Time?** $\delta b^1 + (\delta - 1) b^2 + \ldots + b^\delta = O(b^\delta)$
- **Space?** $O(\delta b)$
- **Optimal?** Yes, if step cost = 1

Properties of IDS

Properties

- **Complete?** Yes
- **Time?** $\delta b^1 + (\delta - 1) b^2 + \ldots + b^\delta = O(b^\delta)$
- **Space?** $O(\delta b)$
- **Optimal?** Yes, if step cost = 1

Properties of IDS

Properties

- **Complete?** Yes
- **Time?** $\delta b^1 + (\delta - 1)b^2 + \ldots + b^\delta = O(b^\delta)$
- **Space?** $O(\delta b)$
- **Optimal?** Yes, if step cost = 1

Properties of IDS

Properties

- **Complete?** Yes
- **Time?** $\delta b^1 + (\delta - 1)b^2 + \ldots + b^\delta = O(b^\delta)$
- **Space?** $O\left(\delta b\right)$
- **Optimal?** Yes, if step cost $= 1$

Iterative Deepening Search Works

Setup - Thanks to Felipe 2015 Class

- D_k the search depth in the algorithm at step k in the wrap part of the algorithm
  - Which can have certain step size!!!

Iterative Deepening Search Works

Theorem (IDS works)

Let $d_{\min}$min the minimum depth of all goal states in the search tree rooted at s. Suppose that

$$D_{k-1} < d_{\min} \leq D_k$$

where $D_0 = 0$. Then IDS will find a goal whose depth is as much D_k.

Iterative Deepening Search Works

Theorem (IDS works)

Let $d_{\min}$min the minimum depth of all goal states in the search tree rooted at s. Suppose that

$$D_{k-1} < d_{\min} \leq D_k$$

where $D_0 = 0$. Then IDS will find a goal whose depth is as much D_k.

Cinvestav

Proof

Since $b > 0$ and finite

We know that the algorithm Depth-Limited Search has no vertices below depth D making the tree finite

In addition

A dept-first search will find the solution in such a tree if any exist.

By definition of $d_{\min}$

The tree generated by Depth-Limited Search must have a goal if and only if $D \geq d_{\min}$.

Proof

Since $b > 0$ and finite

We know that the algorithm Depth-Limited Search has no vertices below depth D making the tree finite

In addition

A dept-first search will find the solution in such a tree if any exist.

By definition of $d_{\min}$

The tree generated by Depth-Limited Search must have a goal if and only if $D \geq d_{\min}$.

Proof

Since $b > 0$ and finite

We know that the algorithm Depth-Limited Search has no vertices below depth D making the tree finite

In addition

A dept-first search will find the solution in such a tree if any exist.

By definition of $d_{\min}$

The tree generated by Depth-Limited Search must have a goal if and only if $D \geq d_{\min}$.

Cinvestav

Proof

Thus

No goal can be find until $D = D_k$ at which time a goal will be found.

Because

The Goal is in the tree, its depth is at most D_k.

Proof

Thus

No goal can be find until $D = D_k$ at which time a goal will be found.

Because

The Goal is in the tree, its depth is at most D_k.

Iterative Deepening Search Problems

Theorem (Upper Bound of calls to IDS)

- Suppose that $D_k = k$ and $b > 1$ (Branching greater than one) for all non-goal vertices s. Let be I the number of calls to Depth Limited Search until a solution is found. Let L be the number of vertices placed in the queue by the BFS. Then, $I < 3\,(L + 1)$.

Note

- The theorem points that at least that IDS will be called at most 3 times the number of vertices placed in the queue by BFS.

Iterative Deepening Search Problems

Theorem (Upper Bound of calls to IDS)

- Suppose that $D_k = k$ and $b > 1$ (Branching greater than one) for all non-goal vertices s. Let be I the number of calls to Depth Limited Search until a solution is found. Let L be the number of vertices placed in the queue by the BFS. Then, $I < 3(L+1)$.

Note

- The theorem points that at least that IDS will be called at most 3 times the number of vertices placed in the queue by BFS.

Proof

Claim

- Suppose $b > 1$ for any non-goal vertex. Let κ be the least depth of any goal.
- Let d_k be the number of vertices in the search tree at depth k.
- Let m_k be the number of vertices at depth less than k.

Cinvestav

Proof

Claim

- Suppose $b > 1$ for any non-goal vertex. Let κ be the least depth of any goal.
- Let d_k be the number of vertices in the search tree at depth k.
- Let m_k be the number of vertices at depth less than k.

Thus, for $k \leq \kappa$

We have that

- $m_k < d_k$
- $m_{k-1} \leq \frac{1}{2} m_k$

Proof

Claim

- Suppose $b > 1$ for any non-goal vertex. Let κ be the least depth of any goal.
- Let d_k be the number of vertices in the search tree at depth k.
- Let m_k be the number of vertices at depth less than k.

Thus, for $k \leq \kappa$

We have that

- $m_k < d_k$
- $m_{k-1} \leq \frac{1}{2} m_k$

Proof

Claim

- Suppose $b > 1$ for any non-goal vertex. Let κ be the least depth of any goal.
- Let d_k be the number of vertices in the search tree at depth k.
- Let m_k be the number of vertices at depth less than k.

Thus, for $k \leq \kappa$

We have that

- $m_k < d_k$
- $m_{k-1} \leq \frac{1}{2} m_k$

Proof

Claim

- Suppose $b > 1$ for any non-goal vertex. Let κ be the least depth of any goal.
- Let d_k be the number of vertices in the search tree at depth k.
- Let m_k be the number of vertices at depth less than k.

Thus, for $k \leq \kappa$

We have that

- $m_k < d_k$
- $m_{k-1} \leq \frac{1}{2} m_k$

Proof

Claim

- Suppose $b > 1$ for any non-goal vertex. Let κ be the least depth of any goal.
- Let d_k be the number of vertices in the search tree at depth k.
- Let m_k be the number of vertices at depth less than k.

Thus, for $k \leq \kappa$

We have that

- $m_k < d_k$
- $m_{k-1} \leq \frac{1}{2} m_k$

Cinvestav

Proof

Claim

- Suppose $b > 1$ for any non-goal vertex. Let κ be the least depth of any goal.
- Let d_k be the number of vertices in the search tree at depth k.
- Let m_k be the number of vertices at depth less than k.

Thus, for $k \leq \kappa$

We have that

- $m_k < d_k$
- $m_{k-1} \leq \frac{1}{2} m_k$

Proof

Thus, we have

$$m_k \leq \frac{1}{2} m_{k+1} \leq \left(\frac{1}{2}\right)^2 m_{k+1} \leq \ldots \leq \left(\frac{1}{2}\right)^{\kappa-k} m_\kappa \tag{1}$$

Suppose

The first goal encountered by the BFS is the n^{th} vertex at depth κ.

We have that

$$L = m_\kappa + n - 1 \tag{2}$$

because the goal is not placed on the queue of the BFS.

Cinvestav

Proof

Thus, we have

$$m_k \leq \frac{1}{2} m_{k+1} \leq \left(\frac{1}{2}\right)^2 m_{k+1} \leq \ldots \leq \left(\frac{1}{2}\right)^{\kappa - k} m_\kappa \tag{1}$$

Suppose

The first goal encountered by the BFS is the n^{th} vertex at depth κ .

We have that

$$L = m_\kappa + n - 1 \tag{2}$$

because the goal is not placed on the queue of the BFS.

Proof

Thus, we have

$$m_k \leq \frac{1}{2} m_{k+1} \leq \left(\frac{1}{2}\right)^2 m_{k+1} \leq ... \leq \left(\frac{1}{2}\right)^{\kappa - k} m_\kappa \quad (1)$$

Suppose

The first goal encountered by the BFS is the n^{th} vertex at depth κ .

We have that

$$L = m_\kappa + n - 1 \quad (2)$$

because the goal is not placed on the queue of the BFS.

Proof

The total number of call of DLS

For $D_k = k < \kappa$ is

$$m_k + d_k \tag{3}$$

Proof

The total number of calls of DLS before we find the solution

$$
\begin{aligned}
I &= \sum_{k=0}^{\kappa-1} \left[m_k + d_k\right] + m_\kappa + n \\
&= \sum_{k=0}^{\kappa-1} m_{k+1} + m_\kappa + n \\
&= \sum_{k=1}^{\kappa} m_k + m_\kappa + n \\
&\leq \sum_{k=1}^{\kappa} \left(\frac{1}{2}\right)^{\kappa-k} m_\kappa + m_\kappa + n \\
&< m_\kappa \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^{i} + m_\kappa + n \\
&< 2m_\kappa + m_\kappa + n = 2m_\kappa + L + 1 \leq 3\left(L+1\right)
\end{aligned}
$$

Proof

The total number of calls of DLS before we find the solution

$$
\begin{aligned}
I &= \sum_{k=0}^{\kappa-1} [m_k + d_k] + m_\kappa + n \\
&= \sum_{k=0}^{\kappa-1} m_{k+1} + m_\kappa + n \\
&= \sum_{k=1}^{\kappa} m_k + m_\kappa + n \\
&\leq \sum_{k=1}^{\kappa} \left(\frac{1}{2}\right)^{\kappa-k} m_\kappa + m_\kappa + n \\
&< m_\kappa \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^{i} + m_\kappa + n \\
&< 2m_\kappa + m_\kappa + n = 2m_\kappa + L + 1 \leq 3\,(L+1)
\end{aligned}
$$

Proof

The total number of calls of DLS before we find the solution

$$
\begin{aligned}
I &= \sum_{k=0}^{\kappa-1} \left[m_k + d_k \right] + m_\kappa + n \\
&= \sum_{k=0}^{\kappa-1} m_{k+1} + m_\kappa + n \\
&= \sum_{k=1}^{\kappa} m_k + m_\kappa + n \\
&\leq \sum_{k=1}^{\kappa} \left(\frac{1}{2} \right)^{\kappa-k} m_\kappa + m_\kappa + n \\
&< m_\kappa \sum_{i=0}^{\infty} \left(\frac{1}{2} \right)^{i} + m_\kappa + n \\
&< 2m_\kappa + m_\kappa + n = 2m_\kappa + L + 1 \leq 3\left(L + 1 \right)
\end{aligned}
$$

Proof

The total number of calls of DLS before we find the solution

$$
\begin{aligned}
I &= \sum_{k=0}^{\kappa-1} \left[m_k + d_k \right] + m_\kappa + n \\
&= \sum_{k=0}^{\kappa-1} m_{k+1} + m_\kappa + n \\
&= \sum_{k=1}^{\kappa} m_k + m_\kappa + n \\
&\leq \sum_{k=1}^{\kappa} \left(\frac{1}{2} \right)^{\kappa-k} m_\kappa + m_\kappa + n \\
&< m_\kappa \sum_{i=0}^{\infty} \left(\frac{1}{2} \right)^{i} + m_\kappa + n \\
&< 2m_\kappa + m_\kappa + n = 2m_\kappa + L + 1 \leq 3\left(L+1\right)
\end{aligned}
$$

Proof

The total number of calls of DLS before we find the solution

$$
\begin{aligned}
I &= \sum_{k=0}^{\kappa-1} \left[m_k + d_k \right] + m_\kappa + n \\
&= \sum_{k=0}^{\kappa-1} m_{k+1} + m_\kappa + n \\
&= \sum_{k=1}^{\kappa} m_k + m_\kappa + n \\
&\leq \sum_{k=1}^{\kappa} \left(\frac{1}{2} \right)^{\kappa-k} m_\kappa + m_\kappa + n \\
&< m_\kappa \sum_{i=0}^{\infty} \left(\frac{1}{2} \right)^{i} + m_\kappa + n \\
&< 2m_\kappa + m_\kappa + n = 2m_\kappa + L + 1 \leq 3\left(L + 1\right)
\end{aligned}
$$

Proof

The total number of calls of DLS before we find the solution

$$
\begin{aligned}
I &= \sum_{k=0}^{\kappa-1} [m_k + d_k] + m_\kappa + n \\
&= \sum_{k=0}^{\kappa-1} m_{k+1} + m_\kappa + n \\
&= \sum_{k=1}^{\kappa} m_k + m_\kappa + n \\
&\leq \sum_{k=1}^{\kappa} \left(\frac{1}{2}\right)^{\kappa-k} m_\kappa + m_\kappa + n \\
&< m_\kappa \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^{i} + m_\kappa + n \\
&< 2m_\kappa + m_\kappa + n = 2m_\kappa + L + 1 \leq 3\,(L+1)
\end{aligned}
$$

Outline

1 Motivation
- Mimicking the way Human Solve Problems
- What is Search?

2 First Idea, State Space Problem
- Introduction
- Better Representation
  - Example
- Solution Definition
- Weighted State Space Problem
- Evaluation of Search Strategies
- Sparse Representation of Graphs

3 Uninformed Graph Search Algorithms
- Implicit State Space Graph
- Back to Implicit State Space Definition
- Basic Functions
- Depth-First Search
- Breadth-First Search
- Combining DFS and BFS
- We Have the Results of Solving a Maze

4 Different ways of doing Stuff
- What happened when you have weights?
- What to do with negative weights?
- Implicit Bellman-Ford

In the Class of 2014

The Class of 2014

- They solved a maze using the previous techniques using Python as base language.

The Maze was Randomly Generated

- Using a Randomize Prim Algorithm

Here is important to notice

- The Problem is the number of nodes explored each time

Cinvestav

In the Class of 2014

The Class of 2014

- They solved a maze using the previous techniques using Python as base language.

The Maze was Randomly Generated

- Using a Randomize Prim Algorithm

Here is important to notice

- The Problem is the number of nodes explored each time

In the Class of 2014

The Class of 2014

- They solved a maze using the previous techniques using Python as base language.

The Maze was Randomly Generated

- Using a Randomize Prim Algorithm

Here is important to notice

- The Problem is the number of nodes explored each time

Table Maze Example

Thanks to Lea and Orlando Class 2014 Cinvestav

Size of Maze	**40×20**			
Start	$(36, 2)$			
Goal	$(33, 7)$			
Algorithm	**Expanded Nodes**	**Generated Nodes**	**Path Size**	**#Iterations**
DFS	**482**	**502**	**35**	**NA**
BFS	**41**	**47**	**9**	**NA**
IDS	**1090**	**3197**	**9**	**9**
IDA*	**11**	**20**	**9**	**2**

Outline

1 Motivation
- Mimicking the way Human Solve Problems
- What is Search?

2 First Idea, State Space Problem
- Introduction
- Better Representation
  - Example
- Solution Definition
- Weighted State Space Problem
- Evaluation of Search Strategies
- Sparse Representation of Graphs

3 Uninformed Graph Search Algorithms
- Implicit State Space Graph
- Back to Implicit State Space Definition
- Basic Functions
- Depth-First Search
- Breadth-First Search
- Combining DFS and BFS
- We Have the Results of Solving a Maze

4 Different ways of doing Stuff
- What happened when you have weights?
- What to do with negative weights?
- Implicit Bellman-Ford

Cinvestav

Weights in the Implicit Graph

Wights in a Graph

- Until now, we have been looking to implicit graphs without weights.
- What to do if we have a function $w : E \rightarrow \mathbb{R}$ such that there is a variability in expanding each path!!!

Cinvestav

Weights in the Implicit Graph

Wights in a Graph

- Until now, we have been looking to implicit graphs without weights.
- What to do if we have a function $w : E \rightarrow \mathbb{R}$ such that there is a variability in expanding each path!!!

Algorithms to attack the problem

- Dijkstra's Algorithm
- Bellman-Ford Algorithm

Weights in the Implicit Graph

Wights in a Graph

- Until now, we have been looking to implicit graphs without weights.
- What to do if we have a function $w : E \to \mathbb{R}$ such that there is a variability in expanding each path!!!

Algorithms to attack the problem

- Dijkstra's Algorithm
- Bellman-Ford Algorithm

Weights in the Implicit Graph

Wights in a Graph

- Until now, we have been looking to implicit graphs without weights.
- What to do if we have a function $w : E \rightarrow \mathbb{R}$ such that there is a variability in expanding each path!!!

Algorithms to attack the problem

- Dijkstra's Algorithm
- Bellman-Ford Algorithm

Cinvestav

Clearly somethings need to be taken into account!!!

Implementation

- Open List uses a Queue
  - MIN Queue Q == GRAY
  - Out of the Queue Q == BLACK
  - Update == Relax

Dijkstra's algorithm

DIJKSTRA(s, w)

$Open$ **is a MIN queue**
$Closed$ **is a set**
Open.enqueue(s**)**
$Closed = \emptyset$
while $Open \neq \emptyset$
    $u =$**Extract-Min**(Q)
    if $Closed \neq Closed \cup (u)$
        $succ(u) = Expand(u)$
        for each vertex $v \in succ(u)$
            if $Closed \neq Closed \cup (v)$
                Relax(u, v, w)
    $Closed = Closed \cup \{u\}$

Relax Procedure

Basic Algorithm

Procedure Relax(u, v, w)

Input: Nodes u, v and v successor of u

SideEffects: Update parent of v, distance to origin $f(v)$, $Open$ and $Closed$

if $(v \in Open)$ ⇒ **Node generated but not expanded**
    if $(f(u) + w(u, v) < f(v))$
        $parent(v) = u$
        $f(v) = f(u) + w(u, v)$
else
    if $(v \notin Closed)$ ⇒ **Not yet expanded**
        $parent(v) = u$
        $f(v) = f(u) + w(u, v)$
        Insert v into $Open$ with $f(v)$

Complexity

Worst Case Performance - Time Complexity

$$O\left(E + V\log V\right) \tag{4}$$

Space Complexity

$$O\left(V^2\right) \tag{5}$$

Complexity

Worst Case Performance - Time Complexity

$$O\left(E + V\log V\right) \quad (4)$$

Space Complexity

$$O\left(V^2\right) \quad (5)$$

Cinvestav

Correctness Dijkstra's Algorithm

Theorem (Optimality of Dijkstra's)

- In weighted graphs with nonnegative weight function the algorithm of Dijkstra's algorithm is optimal.

Theorem (Correctness of Dijkstra's)

- If the weight function w of a problem graph $G = (V, E, w)$ is strictly positive and if the weight of every infinite path is infinite, then Dijkstra's algorithm terminates with an optimal solution.

Correctness Dijkstra's Algorithm

This was shown in the previous class

- Analysis of Algorithms...

Outline

1. Motivation
   - Mimicking the way Human Solve Problems
   - What is Search?
2. First Idea, State Space Problem
   - Introduction
   - Better Representation
     - Example
   - Solution Definition
   - Weighted State Space Problem
   - Evaluation of Search Strategies
   - Sparse Representation of Graphs
3. Uninformed Graph Search Algorithms
   - Implicit State Space Graph
   - Back to Implicit State Space Definition
   - Basic Functions
   - Depth-First Search
   - Breadth-First Search
   - Combining DFS and BFS
   - We Have the Results of Solving a Maze
4. Different ways of doing Stuff
   - What happened when you have weights?
   - What to do with negative weights?
   - Implicit Bellman-Ford

Cinvestav

When Negative Weights Exist

Solution

- You can use the Bellman-Ford Algorithm - Basically Dynamic Programming

Bellman uses node relaxation

- $f(v) \leftarrow \min\{f(v), f(u) + w(u, v)\}$

Implementation on an Implicit Graph

- Open List uses a Queue
  - Insert = Enqueue
  - Select = Denqueue

When Negative Weights Exist

Solution

- You can use the Bellman-Ford Algorithm - Basically Dynamic Programming

Bellman uses node relaxation

- $f(v) \leftarrow \min\{f(v), f(u) + w(u, v)\}$

Implementation on an Implicit Graph

- Open List uses a Queue
  - Insert = Enqueue
  - Select = Denqueue

Cinvestav

When Negative Weights Exist

Solution

- You can use the Bellman-Ford Algorithm - Basically Dynamic Programming

Bellman uses node relaxation

- $f(v) \leftarrow \min\{f(v), f(u) + w(u,v)\}$

Implementation on an Implicit Graph

- Open List uses a Queue
  - Insert = Enqueue
  - Select = Denqueue

Outline

1. Motivation
   - Mimicking the way Human Solve Problems
   - What is Search?
2. First Idea, State Space Problem
   - Introduction
   - Better Representation
     - Example
   - Solution Definition
   - Weighted State Space Problem
   - Evaluation of Search Strategies
   - Sparse Representation of Graphs
3. Uninformed Graph Search Algorithms
   - Implicit State Space Graph
   - Back to Implicit State Space Definition
   - Basic Functions
   - Depth-First Search
   - Breadth-First Search
   - Combining DFS and BFS
   - We Have the Results of Solving a Maze
4. Different ways of doing Stuff
   - What happened when you have weights?
   - What to do with negative weights?
   - Implicit Bellman-Ford

Implicit Bellman-Ford

Procedure Implicit Bellman-Ford

Input: Start node s, function w, function $Expand$ and function $Goal$

Output: Cheapest path from s to $t \in T$ stored in $f(s)$

1. $Open \leftarrow \{s\}$
2. $f(s) \leftarrow h(s)$
3. **while** $(Open \neq \emptyset)$
4. $u = Open.dequeue()$
5. $Closed = Closed \cup \{u\}$
6. $Succ(u) \leftarrow Expand(u)$
7.   **for each** $v \in Succ(u)$
8. $improve(u, v)$

Implicit Bellman-Ford

Procedure Implicit Bellman-Ford

Input: Start node s, function w, function $Expand$ and function $Goal$

Output: Cheapest path from s to $t \in T$ stored in $f(s)$

```
Open ← {s}
f(s) ← h(s)
while (Open ≠ ∅)
        u = Open.dequeue()
        Closed = Closed ∪ {u}
        Succ(u) ← Expand(u)
        for each v ∈ Succ(u)
                improve(u, v)
```

Implicit Bellman-Ford

Procedure Implicit Bellman-Ford

Input: Start node s, function w, function $Expand$ and function $Goal$

Output: Cheapest path from s to $t \in T$ stored in $f(s)$

```
Open ← {s}
f(s) ← h(s)
while (Open ≠ ∅)
        u = Open.dequeue()
        Closed = Closed ∪ {u}
        Succ(u) ← Expand(u)
        for each v ∈ Succ(u)
                improve(u, v)
```

Implicit Bellman-Ford

Procedure Implicit Bellman-Ford

Input: Start node s, function w, function $Expand$ and function $Goal$

Output: Cheapest path from s to $t \in T$ stored in $f(s)$

$Open \leftarrow \{s\}$
$f(s) \leftarrow h(s)$
while $(Open \neq \emptyset)$
    $u = Open.dequeue()$
    $Closed = Closed \cup \{u\}$
    $Succ(u) \leftarrow Expand(u)$
    for each $v \in Succ(u)$
        $improve(u, v)$

Implicit Bellman-Ford

Procedure Implicit Bellman-Ford

Input: Start node s, function w, function $Expand$ and function $Goal$

Output: Cheapest path from s to $t \in T$ stored in $f(s)$

1. $Open \leftarrow \{s\}$
2. $f(s) \leftarrow h(s)$
3. **while** $(Open \neq \emptyset)$
4. $\quad u = Open.dequeue()$
5. $\quad Closed = Closed \cup \{u\}$
6. $\quad Succ(u) \leftarrow Expand(u)$
7. $\quad$ **for each** $v \in Succ(u)$
8. $\quad\quad improve(u, v)$

Algorithm

Procedure Improve

Input: **Nodes u and v, number of problem graph node n**

SideEffects: **Update parent of v, $f(v)$, $Open$ and $Closed$**

1. **if** $(v \in Open)$
2.   **if** $(f(u) + w(u,v) < f(v))$
3.   **if** $(lenght(Path(v)) \geq n-1)$
4.   **exit**
5. $parent(v) \leftarrow u$
6.   **Update** $f(v) \leftarrow f(u) + w(u,v)$
7. **else if** $(v \in Closed)$
8.   **if** $(f(u) + w(u,v) < f(v))$
9.   **if** $(lenght(Path(v)) \geq n-1)$
10.   **exit**

Algorithm

Procedure Improve

Input: **Nodes** u **and** v**, number of problem graph node** n

SideEffects: **Update parent of** v**,** $f(v)$**,** $Open$ **and** $Closed$

1. **if** $(v \in Open)$
2. **if** $(f(u) + w(u,v) < f(v))$
3. **if** $(lenght(Path(v)) \geq n-1)$
4. **exit**
5. $parent(v) \leftarrow u$
6. **Update** $f(v) \leftarrow f(u) + w(u,v)$
7. **else if** $(v \in Closed)$
8. **if** $(f(u) + w(u,v) < f(v))$
9. **if** $(lenght(Path(v)) \geq n-1)$
10. **exit**

Cinvestav

Algorithm

Procedure Improve

Input: **Nodes** u **and** v**, number of problem graph node** n

SideEffects: **Update parent of** v**,** $f(v)$**,** $Open$ **and** $Closed$

```
if (v ∈ Open)
    if (f(u) + w(u,v) < f(v))
        if (lenght(Path(v)) ≥ n − 1)
            exit
        parent(v) ← u
        Update f(v) ← f(u) + w(u,v)
else if (v ∈ Closed)
    if (f(u) + w(u,v) < f(v))
        if (lenght(Path(v)) ≥ n − 1)
            exit
```

Cinvestav

Algorithm

Cont...

1. $parent\,(v) \leftarrow u$
2. **Remove** v **from** $Closed$
3. **Update** $f\,(v) \leftarrow f\,(u) + w\,(u, v)$
4. **Enqueue** v **in** $Open$
5. **else**
6. $parent\,(v) \leftarrow u$
7. **Initialize** $f\,(v) \leftarrow f\,(u) + w\,(u, v)$
8. **Enqueue** v **in** $Open$

Algorithm

Cont...

1. $parent\,(v) \leftarrow u$
2. **Remove** v **from** $Closed$
3. **Update** $f\,(v) \leftarrow f\,(u) + w\,(u, v)$
4. **Enqueue** v **in** $Open$
5. **else**
6. $parent\,(v) \leftarrow u$
7. **Initialize** $f\,(v) \leftarrow f\,(u) + w\,(u, v)$
8. **Enqueue** v **in** $Open$

Complexity and Optimality

Theorem (Optimality of Implicit Bellman-Ford)

Implicit Bellman-Ford is correct and computes optimal cost solution paths.

Theorem (Complexity of Implicit Bellman-Ford)

Implicit Bellman-Ford applies no more than $O(VE)$ node generations.

Space Complexity

$$O\left(V^2\right) \tag{6}$$

Complexity and Optimality

Theorem (Optimality of Implicit Bellman-Ford)

Implicit Bellman-Ford is correct and computes optimal cost solution paths.

Theorem (Complexity of Implicit Bellman-Ford)

Implicit Bellman-Ford applies no more than $O(VE)$ node generations.

Space Complexity

$$O\left(V^2\right) \tag{6}$$

Cinvestav

Complexity and Optimality

Theorem (Optimality of Implicit Bellman-Ford)

Implicit Bellman-Ford is correct and computes optimal cost solution paths.

Theorem (Complexity of Implicit Bellman-Ford)

Implicit Bellman-Ford applies no more than $O(VE)$ node generations.

Space Complexity

$$O\left(V^2\right) \tag{6}$$

Cinvestav

Bibliography

S. Edelkamp and S. Schrodl, *Heuristic Search - Theory and Applications.* Academic Press, 2012.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial intelligence*, vol. 27, no. 1, pp. 97–109, 1985.