Uninformed Search of an Explicit Graph Without Costs

Exploring Alternatives With Search

Douglas H. Fisher

Searching an Explicit Graph Without Checking for Repeated Vertices

Function Search (Vertices V, Arcs A, v_0, G)

/* Given:

V is a set of atomic labels representing vertices in a graph

A is a set of directed arcs (aka edges) between two nodes in V

v_0 is a starting vertex, in V

G is a set of goal vertices, each in V

Return:

path of vertices (and arcs) from v_0 to a member of G

Local:

Frontier is a collection of paths */

Frontier = [$\langle v_0 \rangle$]

while Frontier != [ ] do //search dead ends can eventually in an empty the Frontier

select and remove $\langle v_0, ..., v_k \rangle$ from Frontier

if v_k in G then return $\langle v_0, ..., v_k \rangle$

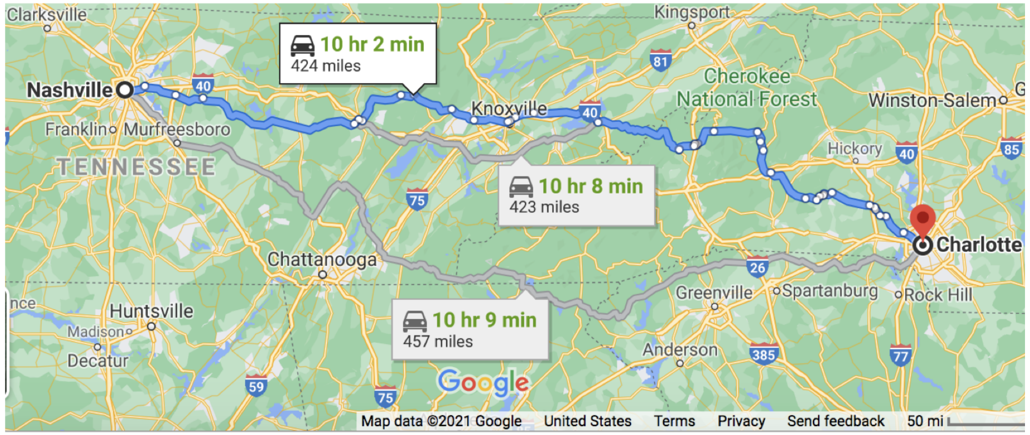
for each v such that (v_k, v) in A

Frontier = Frontier + $\langle v_0, ..., v_k, v \rangle$

return $\langle\rangle$

*A dead end is a vertex from which there are no directed arcs out of the vertex.

Route planning is
abstracted to search an explicit directed graph.

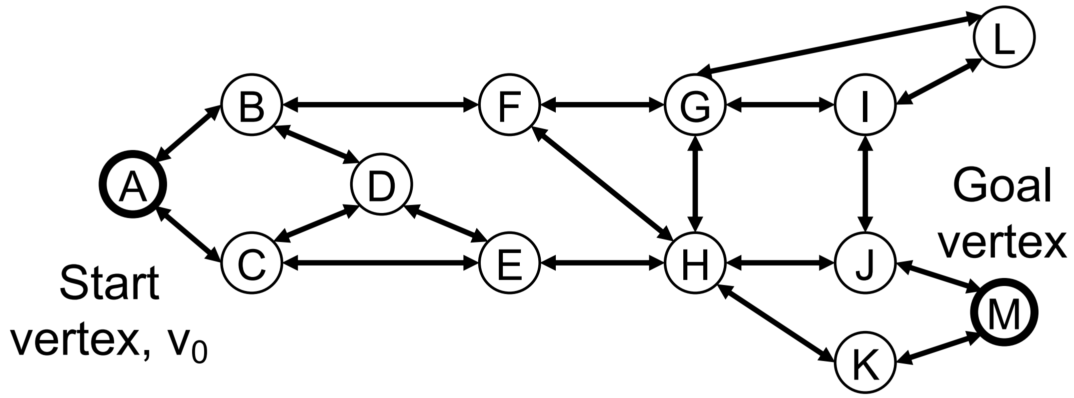


Douglas H. Fisher

Searching an Explicit Graph Without Checking for Repeated Vertices

```
Function Search (Vertices V, Arcs A, v0, G)
/*  Given:
        V is a set of atomic labels representing vertices in a graph
        A is a set of directed arcs (aka edges) between two nodes in V
        v0 is a starting vertex, in V
        G is a set of goal vertices, each in V
    Return:
        path of vertices (and arcs) from v0 to a member of G
    Local:
        Frontier is a collection of paths */
    Frontier = [ 〈v0〉 ]

    while Frontier != [ ] do
        select and remove 〈v0, ..., vk〉 from Frontier
        if vk in G then return 〈v0, ..., vk〉
        for each v such that (vk, v) in A
            Frontier = Frontier + 〈v0, ..., vk, v〉
    return 〈〉
```



$\longleftrightarrow$ is shorthand for $\longleftarrow$ $\longrightarrow$

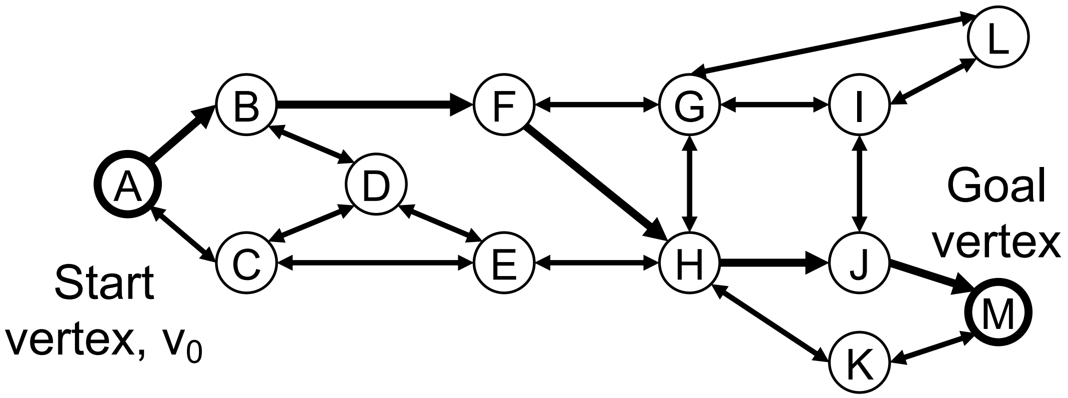
Douglas H. Fisher

Searching an Explicit Graph Without Checking for Repeated Vertices

Function Search (Vertices V, Arcs A, v_0, G)
/* Given:
    V is a set of atomic labels representing vertices in a graph
    A is a set of directed arcs (aka edges) between two nodes in V
    v_0 is a starting vertex, in V
    G is a set of goal vertices, each in V
  Return:
    path of vertices (and arcs) from v_0 to a member of G
  Local:
    Frontier is a collection of paths */
  Frontier = [$\langle v_0 \rangle$]

  while Frontier != [ ] do
    select and remove $\langle v_0, ..., v_k \rangle$ from Frontier
    if v_k in G then return $\langle v_0, ..., v_k \rangle$
    for each v such that (v_k, v) in A
      Frontier = Frontier + $\langle v_0, ..., v_k, v \rangle$
  return $\langle \rangle$

In this case, we would want to return a path to a goal (e.g., **〈(A B) (B F) (F H) (H J) (J M)〉**) rather than just a goal vertex, which we know anyways



Douglas H. Fisher

Searching an Explicit Graph Without Checking for Repeated Vertices

```
Function Search (Vertices V, Arcs A, v0, G)
/*  Given:
        V is a set of atomic labels representing vertices in a graph
        A is a set of directed arcs (aka edges) between two nodes in V
        v0 is a starting vertex, in V
        G is a set of goal vertices, each in V
   Return:
       path of vertices (and arcs) from v0 to a member of G
   Local:
       Frontier is a collection of paths */
   Frontier = [ 〈v0〉 ]

   while Frontier != [ ] do
        select and remove 〈v0, ..., vk〉 from Frontier
        if vk in G then return 〈v0, ..., vk〉
        for each v such that (vk, v) in A
             Frontier = Frontier + 〈v0, ..., vk, v〉
   return 〈〉
```

If Frontier is a stack, then depth-first search
If Frontier is a queue, then breadth-first search

Searching an Explicit Graph Without Checking for Repeated Vertices

Function Search (Vertices V, Arcs A, v_0, G)
/* Given:
      V is a set of atomic labels representing vertices in a graph
      A is a set of directed arcs (aka edges) between two nodes in V
      v_0 is a starting vertex, in V
      G is a set of goal vertices, each in V
   Return:
      path of vertices (and arcs) from v_0 to a member of G
   Local:
      Frontier is a collection of paths */
   Frontier = [$\langle v_0 \rangle$]

while Frontier != [ ] do
   select and remove $\langle v_0, ..., v_k \rangle$ from Frontier
   if v_k in G then return $\langle v_0, ..., v_k \rangle$
   for each v such that (v_k, v) in A
      Frontier = Frontier + $\langle v_0, ..., v_k, v \rangle$
return $\langle\rangle$

$\langle ..., v_k, v \rangle$ is shorthand for $\langle ..., (v_k, v) \rangle$, where (v_k, v) in A.

For example, **$\langle$A B F H J M$\rangle$** is shorthand for **$\langle$(A B) (B F) (F H) (H J) (J M)$\rangle$**

Depth-First Search of an Explicit Graph Without Costs

Exploring Alternatives With Search

Douglas H. Fisher

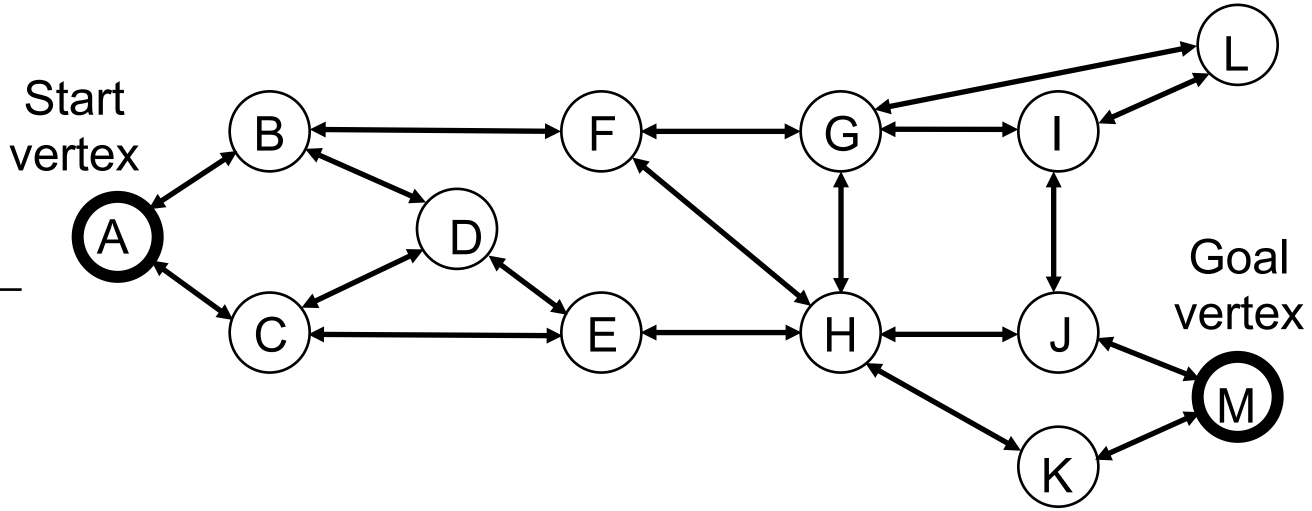
Depth-First Search of a Graph (Without Checking for Repeated Vertices)

Frontier (stack of paths)

1. [**⟨A⟩**]
2. [**⟨A B⟩** **⟨A C ⟩**]
3. [**⟨A B F⟩** **⟨A B D⟩** **⟨A B A⟩** ⟨A C⟩]

*Iteration of **while** loop*

Top of stack (along left)




```
while Frontier != [ ] do
    select and remove ⟨v0, ..., vk⟩ from Frontier
    if vk in G then return ⟨v0, ..., vk⟩
    for each v such that (vk, v) in A
        Frontier = Frontier + ⟨v0, ..., vk, v⟩
```

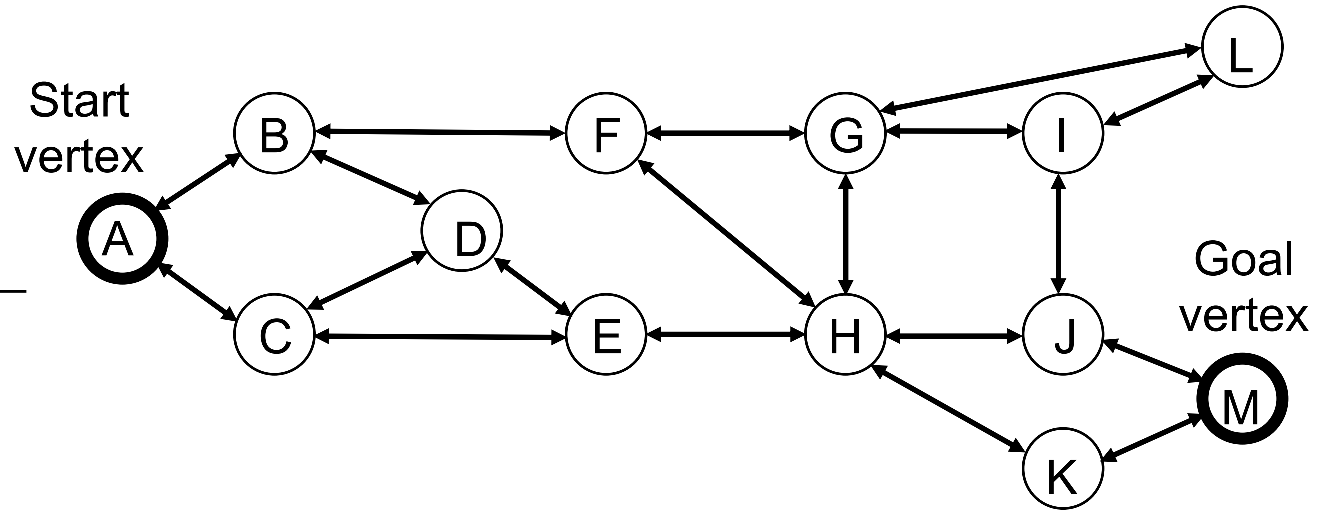
Douglas H. Fisher

Depth-First Search of a Graph (Without Checking for Repeated Vertices)

Frontier (stack of paths)

1. [**⟨A⟩**]
2. [**⟨A $\underline{B}$⟩** **⟨A $\underline{C}$ ⟩**]
3. [**⟨A B $\underline{F}$⟩** **⟨A B $\underline{D}$⟩** **⟨A B $\underline{A}$⟩** ⟨A $\underline{C}$⟩]

```
while Frontier != [ ] do
    select and remove ⟨v_0, ..., v_k⟩ from Frontier
    if v_k in G then return ⟨v_0, ..., v_k⟩
    for each v such that (v_k, v) in A
        Frontier = Frontier + ⟨v_0, ..., v_k, v⟩
```



- Every path begins with start vertex A, v_0
- Last vertex in each path is underlined
- Without checking for repeated vertices, redundant and unnecessarily costly paths can be added to the Frontier
  - ⟨A B $\underline{A}$⟩ is one example

Douglas H. Fisher

Depth-First Search of a Graph (Without Checking for Repeated Vertices)

Frontier (stack of paths)

1. [**〈A〉**]
2. [**〈A B〉**   **〈A C〉**]
3. [**〈A B F〉**   **〈A B D〉**   **〈A B A〉**   〈A C〉]

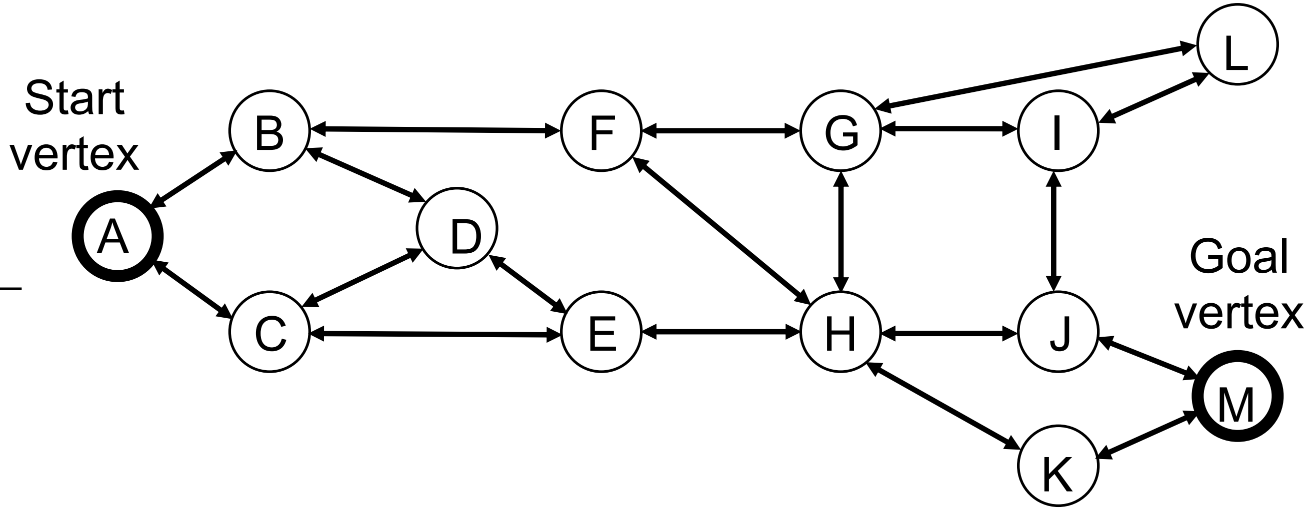
13. while Frontier != [ ] do

14. select and remove 〈v_0, ..., v_k〉 from Frontier

15. if v_k in G then return 〈v_0, ..., v_k〉

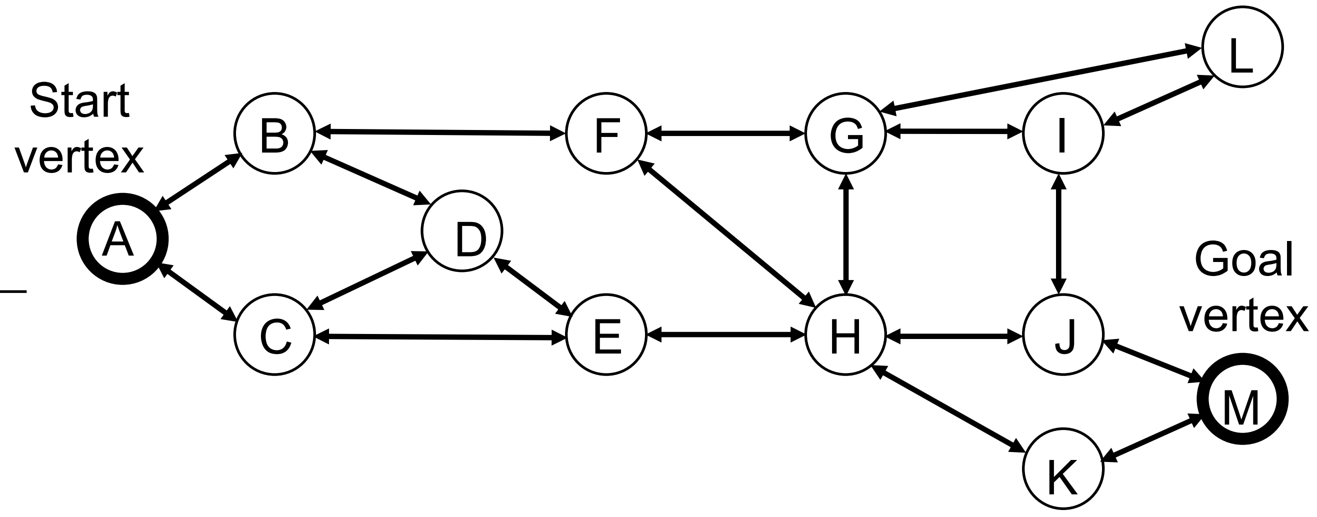
16. for each v such that (v_k, v) in A

17. Frontier = Frontier + 〈v_0, ..., v_k, v〉



- Boldface indicates that a path, such as **〈A B F〉** in step 3, was added to the Frontier in the most recent iteration, when its parent 〈A B〉 in step 2 was removed from the Frontier
- Regular font indicates that a path, such as 〈A C〉 in step 3, was on the previous instance of the Frontier

Depth-First Search of a Graph (Without Checking for Repeated Vertices)



Frontier (stack of paths)

1. [**⟨A⟩**]
2. [**⟨A B⟩** **⟨A C ⟩**]
3. [**⟨A B F⟩** **⟨A B D⟩** **⟨A B A⟩** ⟨A C⟩]

If you wish, pause the video and complete the next iteration or two before continuing.

while Frontier != [ ] do
select and remove ⟨v_0, ..., v_k⟩ from Frontier
if v_k in G then return ⟨v_0, ..., v_k⟩
for each v such that (v_k, v) in A
Frontier = Frontier + ⟨v_0, ..., v_k, v⟩

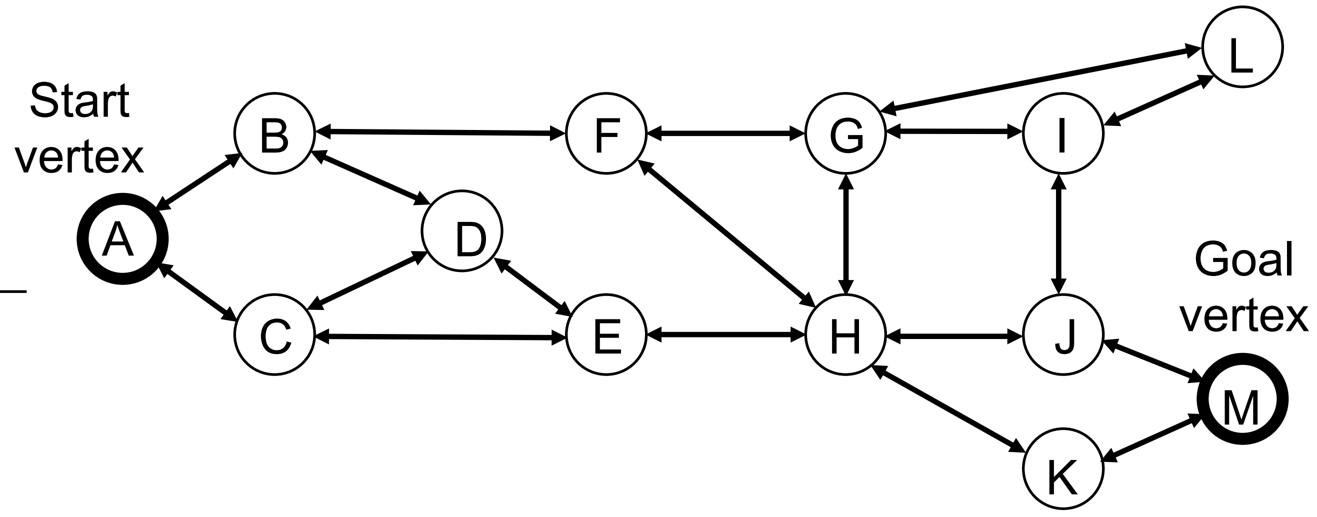
Douglas H. Fisher

Depth-First Search of a Graph (Without Checking for Repeated Vertices)

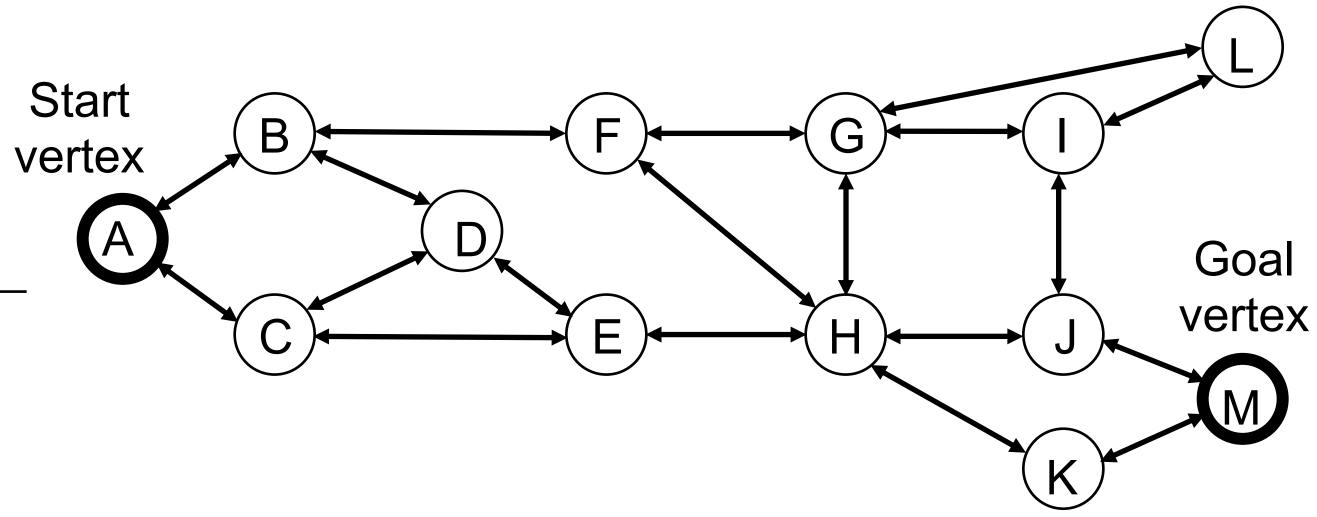
Frontier (stack of paths)

1. [**⟨A⟩**]
2. [**⟨A B⟩** **⟨A C⟩**]
3. [**⟨A B F⟩** **⟨A B D⟩** **⟨A B A⟩** ⟨A C⟩]
4. [**⟨A B F G⟩** **⟨A B F H⟩** **⟨A B F B⟩** ⟨A B D⟩ ⟨A B A⟩ ⟨A C⟩]

Examples of paths with loops



Depth-First Search of a Graph (Without Checking for Repeated Vertices)



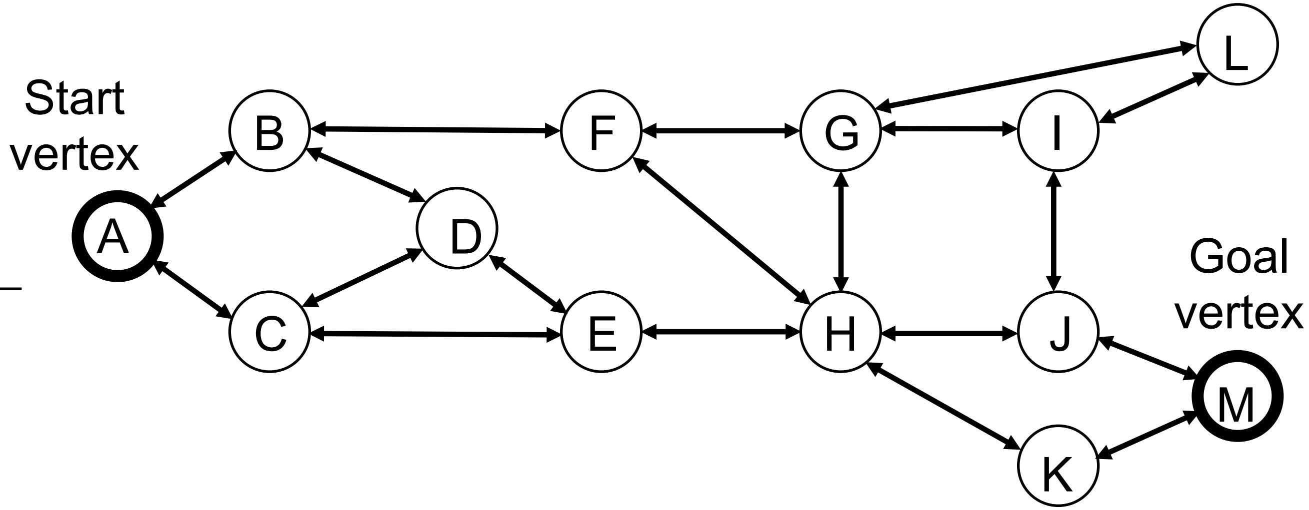
Frontier (stack of paths)

1. [**⟨A⟩**]
2. [**⟨A B⟩** **⟨A C ⟩**]
3. [**⟨A B F⟩** **⟨A B D⟩** **⟨A B A⟩** ⟨A C⟩]
4. [**⟨A B F G⟩** **⟨A B F H⟩** **⟨A B F B⟩** ⟨A B D⟩ ⟨A B A⟩ ⟨A C⟩]
5. [**⟨A B F G L⟩** **⟨A B F G I⟩** **⟨A B F G F⟩** **⟨A B F G H⟩** ⟨A B F H⟩ ⟨A B F B⟩ ⟨A B D⟩ ⟨A B A⟩ ⟨A C⟩]

Example of redundant, nonloop paths
(two different paths to H)

Douglas H. Fisher

Depth-First Search of a Graph (Without Checking for Repeated Vertices)



Frontier (stack of paths)

1. [**⟨A⟩**]
2. [**⟨A B⟩** **⟨A C⟩**]
3. [**⟨A B F⟩** **⟨A B D⟩** **⟨A B A⟩** ⟨A C⟩]
4. [**⟨A B F G⟩** **⟨A B F H⟩** **⟨A B F B⟩** ⟨A B D⟩ ⟨A B A⟩ ⟨A C⟩]
5. [**⟨A B F G L⟩** **⟨A B F G I⟩** **⟨A B F G F⟩** **⟨A B F G H⟩** ⟨A B F H⟩ ⟨A B F B⟩ ⟨A B D⟩ ⟨A B A⟩ ⟨A C⟩]
6. [**⟨A B F G L G⟩** **⟨A B F G L I⟩** ⟨A B F G I⟩ ⟨A B F G F⟩ ⟨A B F G H⟩ ⟨A B F H⟩ ⟨A B F B⟩ ⟨A B D⟩ ⟨A B A⟩ ⟨A C⟩]
7. [**⟨A B F G L G L⟩** **⟨A B F G L G I⟩** **⟨A B F G L G F⟩** ⟨A B F G L I⟩ ⟨A B F G I⟩ ⟨A B F G F⟩ ⟨A B F G H⟩ ⟨A B F H⟩ ⟨A B F B⟩ ⟨A B D⟩ ⟨A B A⟩ ⟨A C⟩]

By now, you should see the problem of redundant paths and the potential for looping, which is particularly problematic with depth-first search because of potential for infinite loops—consider G L G L as an example.

Uninformed Search
With Checks for Repeated Vertices

Exploring Alternatives With Search

Douglas H. Fisher

Searching an Explicit Graph With Checking for Repeated Vertices and Redundant Paths

```
Function Search (Vertices V, Arcs A, v0, G)
/*  … */
  Frontier = [ 〈v0〉 ]
  Reached = { 〈v0〉 }
  while Frontier != [ ] do    //search dead ends, loops, and other redundant paths can result in an empty Frontier
        select and remove 〈v0, ..., vk〉 from Frontier
        if vk in G then return 〈v0, ..., vk〉
        for each v such that (vk, v) in A
              if !exists 〈v0, ..., v〉 in Reached
                        or Cost({ 〈v0, ..., vk, v〉 ) < Cost({ 〈v0, ..., v〉 )
                   Reached = Reached - 〈v0, ..., v〉 + 〈v0, ..., vk, v〉
                   Frontier = Frontier + 〈v0, ..., vk, v〉
  return 〈〉
```

Douglas H. Fisher

Searching an Explicit Graph With Checking for Repeated Vertices and Redundant Paths

Function Search (Vertices V, Arcs A, v_0, G)

/* … */

Frontier = [$\langle v_0 \rangle$]

Reached = { $\langle v_0 \rangle$ }

while Frontier != [ ] do

select and remove $\langle v_0, \ldots, v_k \rangle$ from Frontier

if v_k in G then return $\langle v_0, \ldots, v_k \rangle$

for each v such that (v_k, v) in A

if !exists $\langle v_0, \ldots, v \rangle$ in Reached // if a path to v does not already exist in Reached then add it

or Cost({ $\langle v_0, \ldots, v_k, v \rangle$) < Cost({ $\langle v_0, \ldots, v \rangle$)

Reached = Reached - $\langle v_0, \ldots, v \rangle$ **+ $\langle v_0, \ldots, v_k, v \rangle$ // if $\langle v_0, \ldots, v \rangle$ doesn't exist,**

… **//then Reached - $\langle v_0, \ldots, v \rangle$ is a no-op**

…

Searching an Explicit Graph With Checking for Repeated Vertices and Redundant Paths

```
Function Search (Vertices V, Arcs A, v_0, G)
/*  … */
  Frontier = [ ⟨v_0⟩ ]
  Reached = { ⟨v_0⟩ }
  while Frontier != [ ] do
        select and remove  ⟨v_0, ..., v_k⟩  from Frontier
        if v_k in G then return  ⟨v_0, ..., v_k⟩
        for each v such that (v_k, v) in A
              if !exists  ⟨v_0, ..., v⟩  in Reached
                        or Cost({ ⟨v_0, ..., v_k, v⟩ ) < Cost({ ⟨v_0, ..., v⟩ )      // if a lesser cost path to v is found,
                    Reached = Reached -  ⟨v_0, ..., v⟩   +  ⟨v_0, ..., v_k, v⟩     //then replace old path to v
                    Frontier = Frontier  +  ⟨v_0, ..., v_k, v⟩
  return  ⟨⟩
```

Searching an Explicit Graph With Checking for Repeated Vertices and Redundant Paths

```
Function Search (Vertices V, Arcs A, v_0, G)
/*  … */
  Frontier = [ ⟨v_0⟩ ]
  Reached = { ⟨v_0⟩ }
  while Frontier != [ ] do
      select and remove  ⟨v_0, ..., v_k⟩  from Frontier
      if v_k in G then return  ⟨v_0, ..., v_k⟩
      for each v such that (v_k, v) in A
          if !exists  ⟨v_0, ..., v⟩  in Explored
                  or Cost({ ⟨v_0, ..., v_k, v⟩ ) < Cost({ ⟨v_0, ..., v⟩ )
              Reached = Reached -  ⟨v_0, ..., v⟩   +  ⟨v_0, ..., v_k, v⟩
              Frontier = Frontier  +  ⟨v_0, ..., v_k, v⟩
  return  ⟨⟩
```

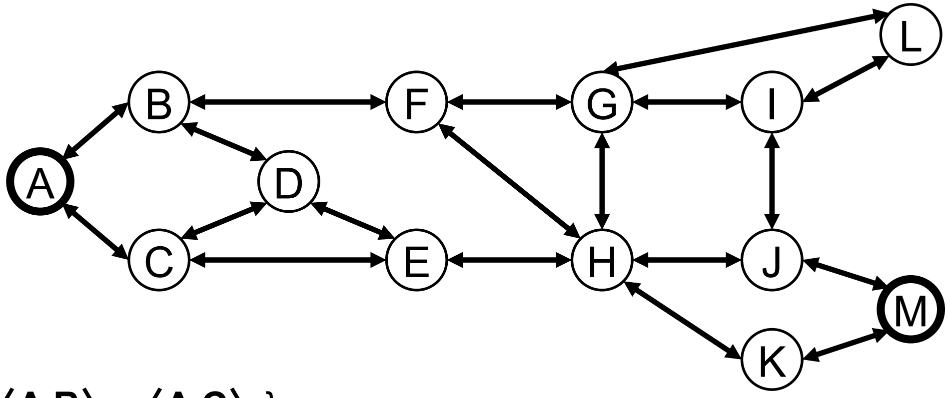
Douglas H. Fisher

Breadth-First Search With Checks for Repeated Vertices

Exploring Alternatives With Search

Douglas H. Fisher

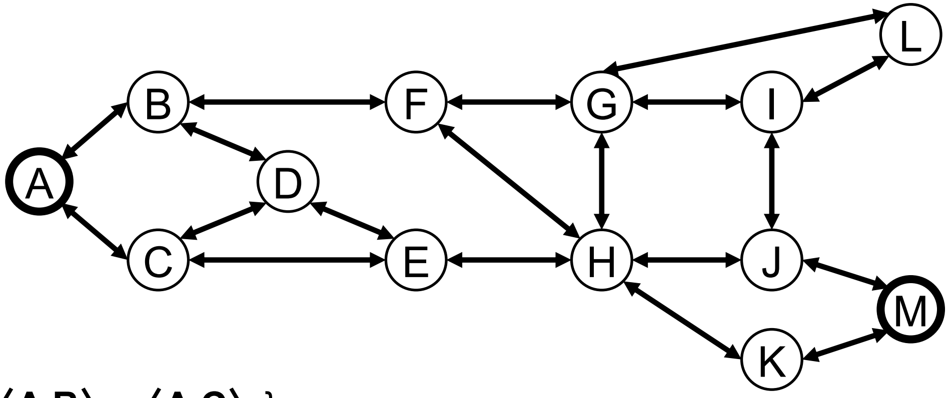
Breadth-First Search of a Graph (With Checking for Repeated Vertices)



Frontier (queue of paths)	Reached
1. [**⟨A⟩**]	1. { **⟨A⟩** }
2. [**⟨A B⟩** **⟨A C⟩**]	2. { ⟨A⟩ **⟨A B⟩** **⟨A C⟩** }

select and remove ⟨v_0, ..., v_k⟩ from Frontier
if v_k in G then return ⟨v_0, ..., v_k⟩
for each v such that (v_k, v) in A
    if !exists ⟨v_0, ..., v⟩ in Reached
        or Cost({ ⟨v_0, ..., v_k, v⟩) < Cost({ ⟨v_0, ..., v⟩)
      Reached = Reached - ⟨v_0, ..., v⟩ + ⟨v_0, ..., v_k, v⟩
      Frontier = Frontier + ⟨v_0, ..., v_k, v⟩

Breadth-First Search of a Graph (With Checking for Repeated Vertices)



Frontier (queue of paths)	Reached
1. [**⟨A⟩**]	1. { **⟨A⟩** }
2. [**⟨A B⟩** **⟨A C⟩**]	2. { ⟨A⟩ **⟨A B⟩** **⟨A C⟩** }
3. [⟨A C⟩ **⟨A B D⟩** **⟨A B F⟩**]	3. { ⟨A⟩ ⟨A B⟩ ⟨A C⟩ **⟨A B D⟩** **⟨A B F⟩** }
4. [⟨A B D⟩ ⟨A B F⟩ ~~**⟨A C D⟩**~~ **⟨A C E⟩**]	4. { ⟨A⟩ ⟨A B⟩ ⟨A C⟩ ⟨A B D⟩ ⟨A B F⟩ **⟨A C E⟩** }

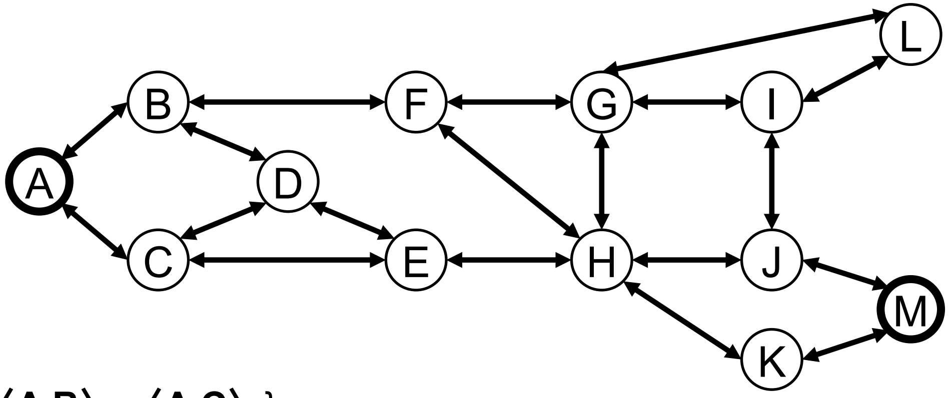
select and remove $\langle v_0, ..., v_k \rangle$ from Frontier
if v_k in G then return $\langle v_0, ..., v_k \rangle$
for each v such that (v_k, v) in A
    if !exists $\langle v_0, ..., v \rangle$ in Reached
        or Cost({ $\langle v_0, ..., v_k, v \rangle$) < Cost({ $\langle v_0, ..., v \rangle$)
      Reached = Reached - $\langle v_0, ..., v \rangle$ + $\langle v_0, ..., v_k, v \rangle$
      Frontier = Frontier + $\langle v_0, ..., v_k, v \rangle$

~~⟨A C D⟩~~ is a redundant, no less costly path to D than ⟨A B D⟩ , and so would not be added to Frontier (or Reached) to begin with. Note that

~~⟨A C D⟩~~ is correctly excluded from Reached already.

Douglas H. Fisher

Breadth-First Search of a Graph (With Checking for Repeated Vertices)



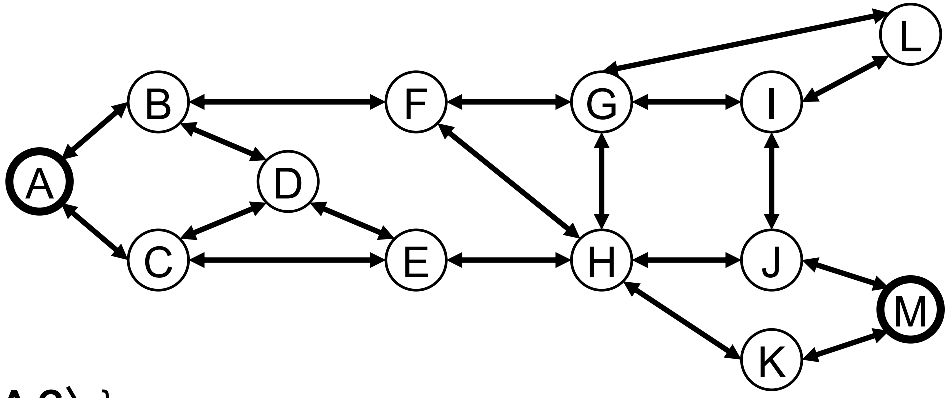
Frontier (queue of paths)	Reached
1. [**⟨A⟩**]	1. { **⟨A⟩** }
2. [**⟨A B⟩** **⟨A C⟩**]	2. { ⟨A⟩ **⟨A B⟩** **⟨A C⟩** }
3. [⟨A C⟩ **⟨A B D⟩** **⟨A B F⟩**]	3. { ⟨A⟩ ⟨A B⟩ ⟨A C⟩ **⟨A B D⟩** **⟨A B F⟩** }
4. [⟨A B D⟩ ⟨A B F⟩ ~~**⟨A C D⟩**~~ **⟨A C E⟩**]	4. { ⟨A⟩ ⟨A B⟩ ⟨A C⟩ ⟨A B D⟩ ⟨A B F⟩ **⟨A C E⟩** }

```
select and remove ⟨v0, ..., vk⟩ from Frontier
if vk in G then return ⟨v0, ..., vk⟩
for each v such that (vk, v) in A
      if !exists ⟨v0, ..., v⟩ in Reached
            or Cost({ ⟨v0, ..., vk, v⟩ ) < Cost({ ⟨v0, ..., v⟩ )
         Reached = Reached - ⟨v0, ..., v⟩ + ⟨v0, ..., vk, v⟩
         Frontier = Frontier + ⟨v0, ..., vk, v⟩
```

If you wish complete the next iteration or two before continuing. The complete breadth-first search is shown on the next slide.

Douglas H. Fisher

Breadth-First Search of a Graph
(With Checking for Repeated Vertices)



Frontier (queue of paths)	Reached
1. [**⟨A⟩**]	1. { **⟨A⟩** }
2. [**⟨A B⟩** **⟨A C⟩**]	2. { ⟨A⟩ **⟨A B⟩** **⟨A C⟩** }
3. [⟨A C⟩ **⟨A B D⟩** **⟨A B F⟩**]	3. { ⟨A⟩ ⟨A B⟩ ⟨A C⟩ **⟨A B D⟩** **⟨A B F⟩** }
4. [⟨A B D⟩ ⟨A B F⟩ **⟨A C E⟩**]	4. { ⟨A⟩ ⟨A B⟩ ⟨A C⟩ ⟨A B D⟩ ⟨A B F⟩ **⟨A C E⟩** }
5. [⟨A B F⟩ ⟨A C E⟩]	5. { ⟨A⟩ ⟨A B⟩ ⟨A C⟩ ⟨A B D⟩ ⟨A B F⟩ ⟨A C E⟩ }
6. [⟨A C E⟩ **⟨A B F G⟩** **⟨A B F H⟩**]	6. { ⟨A⟩ ⟨A B⟩ ⟨A C⟩ ⟨A B D⟩ ⟨A B F⟩ ⟨A C E⟩ **⟨A B F G⟩** **⟨A B F H⟩** }
7. [⟨A B F G⟩ ⟨A B F H⟩]	7. { ⟨A⟩ ⟨A B⟩ … ⟨A B F H⟩ }
8. [⟨A B F H⟩ **⟨A B F G I⟩**]	8. { ⟨A⟩ ⟨A B⟩ … ⟨A B F H⟩ **⟨A B F G I⟩** }
9. [⟨A B F G I⟩ **⟨A B F H J⟩** **⟨A B F H K⟩**]	9. { ⟨A⟩ ⟨A B⟩ … ⟨A B F G I⟩ **⟨A B F H J⟩** **⟨A B F H K⟩** }
10. [⟨A B F H J⟩ **⟨A B F H K⟩** **⟨A B F G I L⟩**]	10. { ⟨A⟩ ⟨A B⟩ … ⟨A B F H K⟩ **⟨A B F G I L⟩** }
11. [⟨A B F H K⟩ ⟨A B F G I L⟩ **⟨A B F H J M⟩**]	11. { ⟨A⟩ ⟨A B⟩ … ⟨A B F G I L⟩ **⟨A B F H J M⟩** }

Finding a goal, M, is two dequeues away.

Douglas H. Fisher

Early Goal Test With Breadth-First Search

while Frontier != [ ] do

select and remove $\langle v_0, \ldots, v_k \rangle$ from Frontier

~~if v_k in G then return $\langle v_0, \ldots, v_k \rangle$~~

No late-goal test after dequeue

for each v such that (v_k, v) in A

if !exists $\langle v_0, \ldots, v \rangle$ in Reached or

~~Cost({ $\langle v_0, \ldots, v_k, v \rangle$ }) < Cost({ $\langle v_0, \ldots, v \rangle$ })~~

Comparing costs not necessary in breadth-first search and paths will automatically be enumerated in order of length

if v in G then return $\langle \mathbf{v_0, \ldots, v_k, v} \rangle$

Rather make early-goal test before enqueue

Reached = Reached - $\langle v_0, \ldots, v \rangle$ + $\langle v_0, \ldots, v_k, v \rangle$

If not goal, then go ahead and enqueue it

Frontier = Frontier + $\langle v_0, \ldots, v_k, v \rangle$

- An early goal test in breadth-first search will still ensure that minimal length paths to goal are found.
- And it is more space- and runtime-efficient than a late goal test in breadth-first search.
- So, we would probably use an early goal test if we knew we would use a breadth-first search, which would be rare, since we would probably use iterative-deepening depth first search (IDDFS) instead (coming up).
- This is a good example that generality, in the form of the generic search algorithm, can be elegant, but not always as efficient when we can make specializing assumptions.

- Question: Can we do an early goal test with IDDFS and still be guaranteed a minimal-length solution?

Early Goal Test With Breadth-First Search (cont.)

while Frontier != [ ] do

  select and remove ⟨v_0, ..., v_k⟩ from Frontier

  ~~if v_k in G then return ⟨v_0, ..., v_k⟩~~

  for each v such that (v_k, v) in A

    if !exists ⟨v_0, ..., v⟩ in Reached or

      ~~Cost({ ⟨v_0, ..., v_k, v⟩ }) < Cost({ ⟨v_0, ..., v⟩ })~~

      if v in G then return **⟨v_0, ..., v_k, v⟩**

      Reached = Reached - ⟨v_0, ..., v⟩ + ⟨v_0, ..., v_k, v⟩

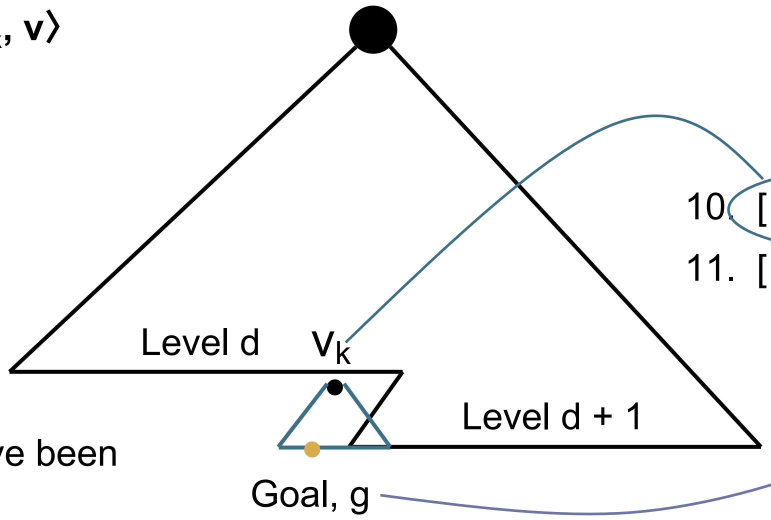
      Frontier = Frontier + ⟨v_0, ..., v_k, v⟩

No late-goal test after dequeue

Comparing costs not necessary in breadth-first search
and paths will automatically be enumerated in order of length

Rather make early-goal test before enqueue

If not goal, then go ahead and enqueue it



10. [⟨A B F H J⟩ ⟨A B F H K⟩ **⟨A B F G I L⟩]**

11. [⟨A B F H K⟩ ⟨A B F G I L⟩]

⟨A B F H J M⟩

How do we know there
is no goal here at level d? →

Because had there been, it would have been
found before its enqueue

Douglas H. Fisher

Iterative Deepening

Exploring Alternatives With Search

Douglas H. Fisher

Iterative Deepening Depth-First Search of a Graph (Without Checking for Repeated Vertices)

Frontier (stack of paths)

1. [**⟨A⟩**] (followed by [ ])

DFS to depth 0

2. [**⟨A⟩**]
3. [**⟨A B⟩** **⟨A C ⟩**]
4. [⟨A C ⟩] (followed by [ ])

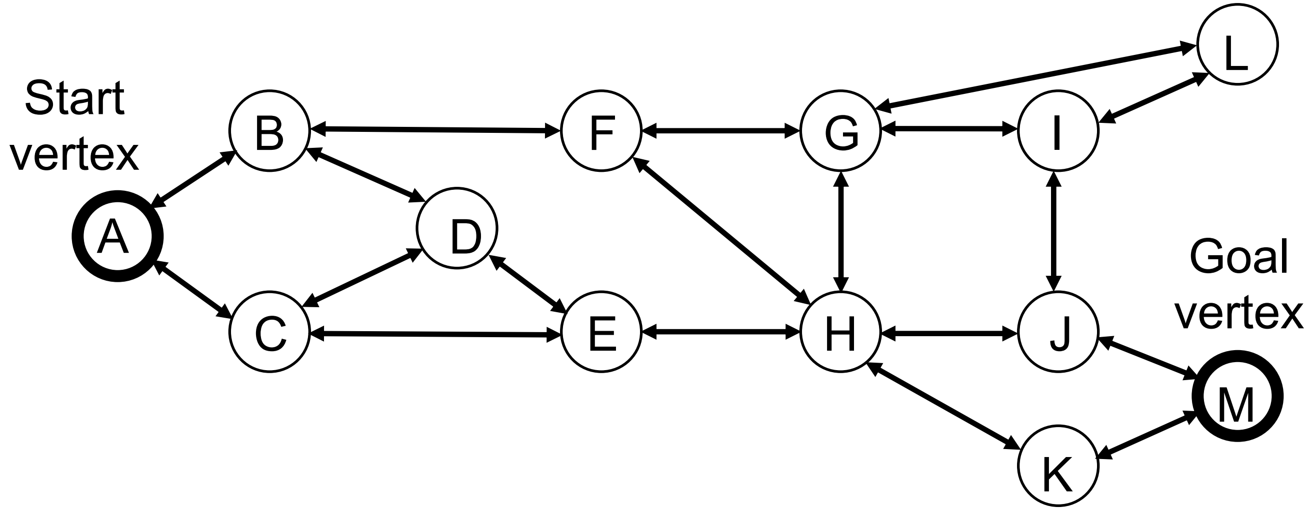
DFS to depth 1

5. [**⟨A⟩**]
6. [**⟨A B⟩** **⟨A C ⟩**]
7. [**⟨A B F⟩** **⟨A B D⟩** **⟨A B A⟩** ⟨A C⟩]
8. [⟨A B D⟩ ⟨A B A⟩ ⟨A C⟩]
9. [⟨A B A⟩ ⟨A C⟩]
10. [⟨A C⟩]
11. [**⟨A C D⟩** **⟨A C E⟩** **⟨A C A⟩**]
12. [⟨A C E⟩ ⟨A C A⟩]
13. [⟨A C A⟩] (followed by [ ])

DFS to depth 2

14. [**⟨A⟩**]

Start DFS to depth 3



IDDFS is preferred over BFS, but why?!?

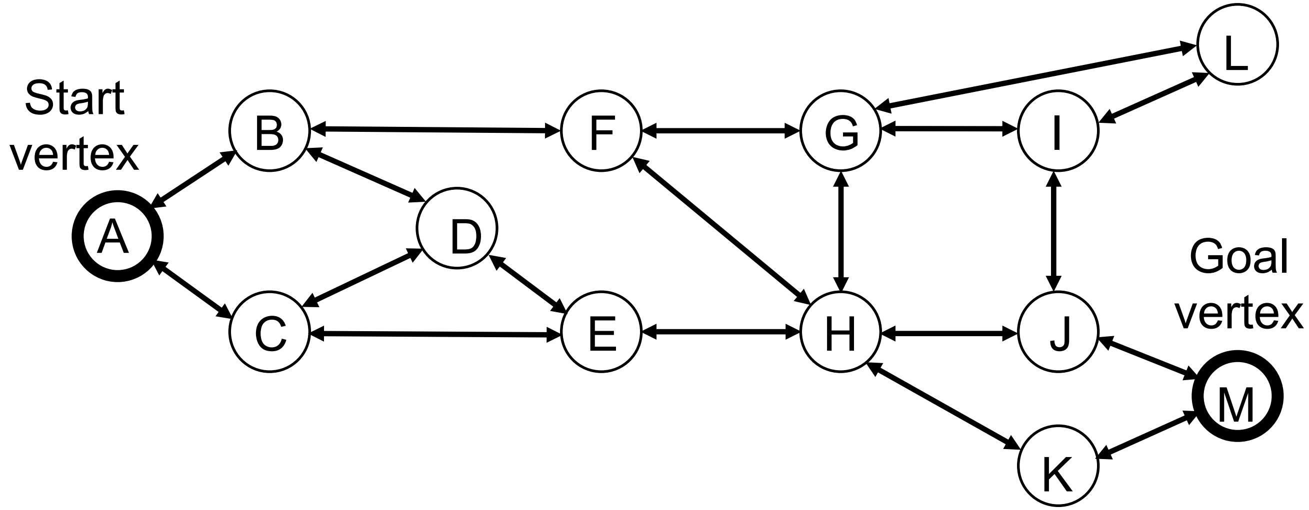
Keep searching to increasing depths until a goal is found. The first goal found is guaranteed to be a shortest path from start to goal.

Douglas H. Fisher

Iterative Deepening Depth-First Search of a Graph (Without Checking for Repeated Vertices)

Frontier (stack of paths)

1. [**⟨A⟩**] (followed by [ ]) — DFS to depth 0
2. [**⟨A⟩**]
3. [**⟨A B⟩** **⟨A C ⟩**]
4. [⟨A C ⟩] (followed by [ ]) — DFS to depth 1
5. [**⟨A⟩**]
6. [**⟨A B⟩** **⟨A C ⟩**]
7. [**⟨A B F⟩** **⟨A B D⟩** **⟨A B A⟩** ⟨A C⟩]
8. [⟨A B D⟩ ⟨A B A⟩ ⟨A C⟩]
9. [⟨A B A⟩ ⟨A C⟩]
10. [⟨A C⟩]
11. [**⟨A C D⟩** **⟨A C E⟩** **⟨A C A⟩**]
12. [⟨A C E⟩ ⟨A C A⟩]
13. [⟨A C A⟩] (followed by [ ]) — DFS to depth 2
14. [**⟨A⟩**] — Start DFS to depth 3



IDDFS is preferred over BFS, but why?!?

- The space requirements of BFS are O(B^d)
  - e.g., with B = 10, and d = 20, O(10^{20}) stuff starts breaking
- The space requirements for IDDFS are O(B*d)
- The runtime cost of BFS is O($\sum_{k=0}^{d} B^k$) = O(B^d), and this is also the runtime cost of IDDFS to depth d! Why?

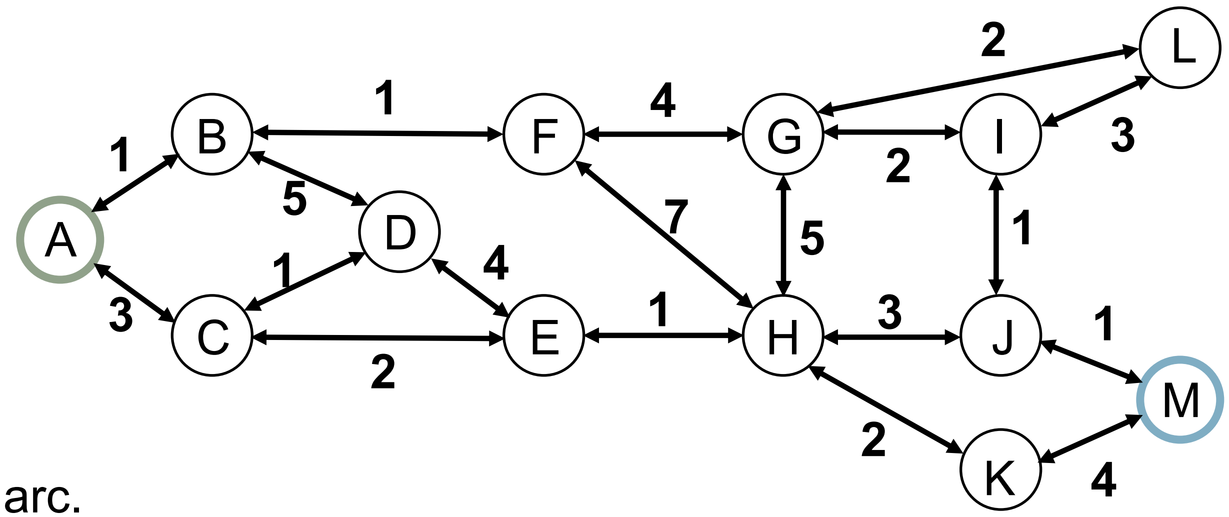
Douglas H. Fisher

Uninformed Search of an Explicit Graph With Costs

Exploring Alternatives With Search

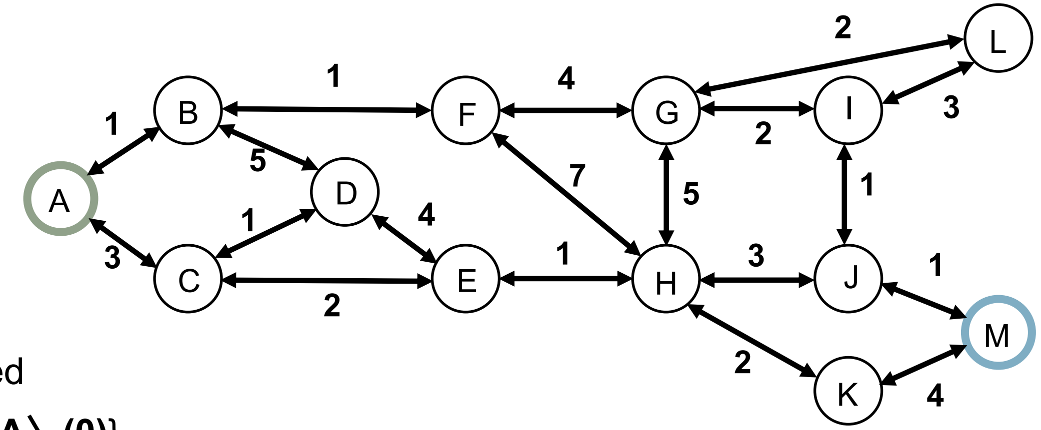
Douglas H. Fisher

Lowest-Cost First Search (aka Uniform Cost Search, aka Dijkstra's Algorithm) of a Graph



- Arc costs label each arc.
- Path costs are the sum of costs on arcs in the path.
  - For example, 〈A B F G〉 (6) has cost 1 + 1 + 4 = 6.
- Double arrow arcs (⟷) is shorthand for two single arrow arcs (⇄) and costs, if any, being equal in both directions.
  - But in many applications, arcs in each direction have different costs (e.g., one direction corresponds to uphill, the other to downhill; one direction is with rush hour traffic, the other is with the lighter flow).
  - But for now, simplifying assumptions apply.

Lowest-Cost First Search (aka Uniform Cost Search, aka Dijkstra's Algorithm) of a Graph (With Checking for Repeated Vertices)



Frontier (priority queue of paths)

1. [**〈A〉 (0)**]
2. [**〈A B〉 (1) 〈A C〉 (3)**]
3. [**〈A B F〉 (2)** 〈A C〉 (3) **〈A B D〉 (6)**]
4. [〈A C〉 (3) 〈A B D〉 (6) **〈A B F G〉 (6) 〈A B F H〉 (9)**]

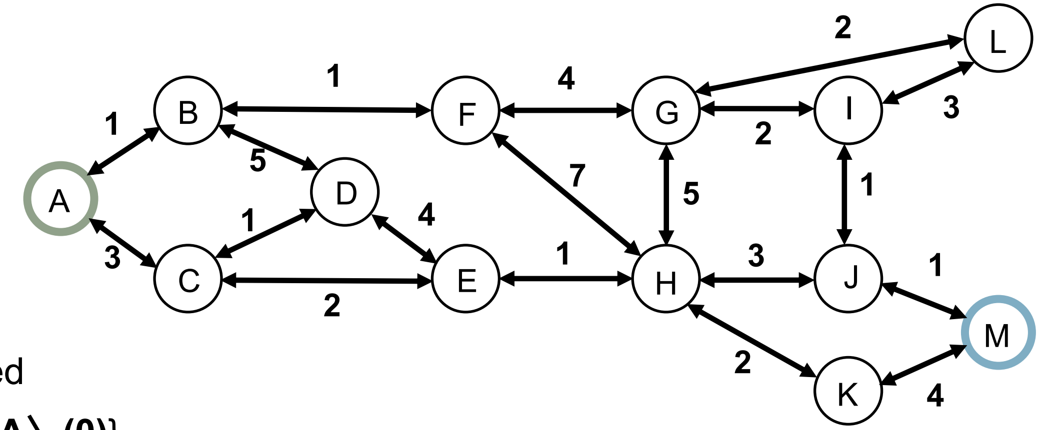
Reached

1. { **〈A〉 (0)**}
2. { 〈A〉 (0) **〈A B〉 (1) 〈A C〉 (3)**}
3. { 〈A〉 (0) 〈A B〉 (1) 〈A C〉 (3) **〈A B F〉 (2) 〈A B D〉 (6)**}
4. { 〈A〉 (0) 〈A B〉 (1) 〈A C〉 (3) 〈A B F〉 (2) 〈A B D〉 (6) **〈A B F G〉 (6) 〈A B F H〉 (9)**}

- Path costs, the sum of costs on arcs in the path, are in parentheses have been added for easy reference.
  - For example, 〈A B F G〉 (6) has cost 1 + 1 + 4 = 6.
- Redundant, more costly paths to a vertex, are not added to Reached or to Frontier.
  - For example, when 〈A B〉 (1) is expanded into 〈A B D〉 (6) and 〈A B F〉 (2), 〈A B A〉 (2) is **not** added since it is a redundant path that is more costly than 〈A〉 (0).

Douglas H. Fisher

Lowest-Cost First Search (aka Uniform Cost Search, aka Dijkstra's Algorithm) of a Graph (With Checking for Repeated Vertices)



Frontier (priority queue of paths)

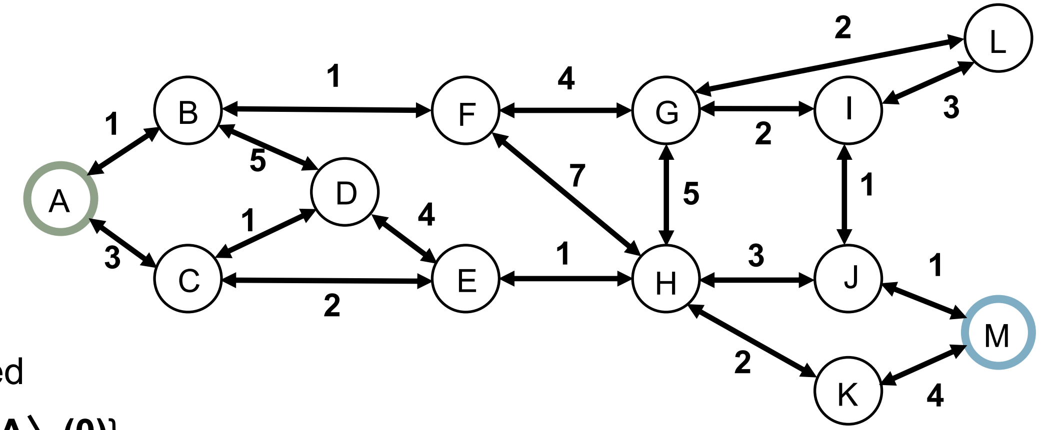
1. [**〈A〉 (0)**]
2. [**〈A B〉 (1) 〈A C〉 (3)**]
3. [**〈A B F〉 (2)** 〈A C〉 (3) **〈A B D〉 (6)**]
4. [〈A C〉 (3) 〈A B D〉 (6) **〈A B F G〉 (6) 〈A B F H〉 (9)**]
5. [**〈A C D〉 (4) 〈A C E〉 (5)** 〈A B D〉 (6) 〈A B F G〉 (6) 〈A B F H〉 (9)]

Reached

1. { **〈A〉 (0)**}
2. { 〈A〉 (0) **〈A B〉 (1) 〈A C〉 (3)**}
3. { 〈A〉 (0) 〈A B〉 (1) 〈A C〉 (3) **〈A B F〉 (2) 〈A B D〉 (6)**}
4. { 〈A〉 (0) 〈A B〉 (1) 〈A C〉 (3) 〈A B F〉 (2) 〈A B D〉 (6) **〈A B F G〉 (6) 〈A B F H〉 (9)**}
5. { 〈A〉 (0) 〈A B〉 (1) 〈A C〉 (3) 〈A B F〉 (2) ~~〈A B D〉 (6)~~ 〈A B F G〉 (6) 〈A B F H〉 (9) **〈A C D〉 (4) 〈A C E〉 (5)**}

The path to D, 〈A B D〉 (6), was added before 〈A C D〉 (4), and the earlier redundant path 〈A B D〉 (6) is removed from Reached, but not from Frontier. Why not Frontier, too?

Lowest-Cost First Search (aka Uniform Cost Search, aka Dijkstra's Algorithm) of a Graph (With Checking for Repeated Vertices)



Frontier (priority queue of paths)

1. [**〈A〉 (0)**]
2. [**〈A B〉 (1) 〈A C〉 (3)**]
3. [**〈A B F〉 (2)** 〈A C〉 (3) **〈A B D〉 (6)**]
4. [〈A C〉 (3) 〈A B D〉 (6) **〈A B F G〉 (6) 〈A B F H〉 (9)**]
5. [**〈A C D〉 (4) 〈A C E〉 (5)** 〈A B D〉 (6) 〈A B F G〉 (6) 〈A B F H〉 (9)]

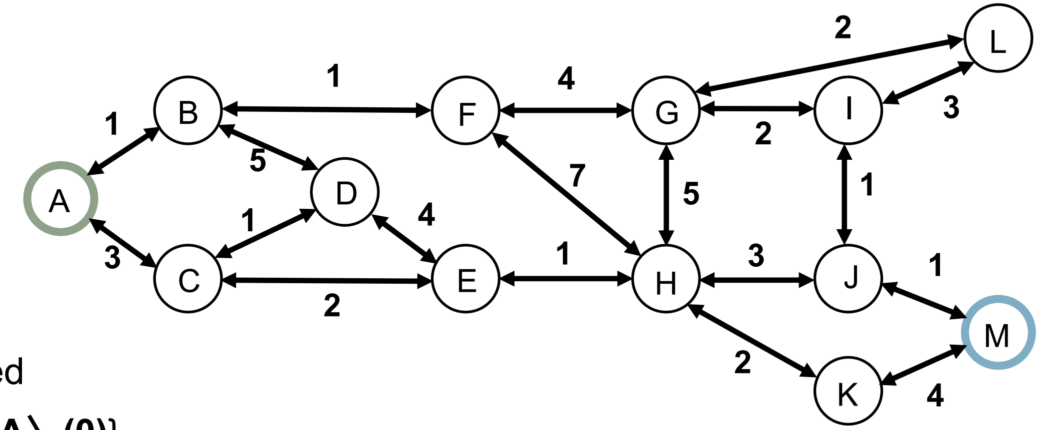
Reached

1. { **〈A〉 (0)**}
2. { 〈A〉 (0) **〈A B〉 (1) 〈A C〉 (3)**}
3. { 〈A〉 (0) 〈A B〉 (1) 〈A C〉 (3) **〈A B F〉 (2) 〈A B D〉 (6)**}
4. { 〈A〉 (0) 〈A B〉 (1) 〈A C〉 (3) 〈A B F〉 (2) 〈A B D〉 (6) **〈A B F G〉 (6) 〈A B F H〉 (9)**}
5. { 〈A〉 (0) 〈A B〉 (1) 〈A C〉 (3) 〈A B F〉 (2) ~~〈A B D〉 (6)~~ 〈A B F G〉 (6) 〈A B F H〉 (9) **〈A C D〉 (4) 〈A C E〉 (5)**}

The path to D, 〈A B D〉 (6), was added before 〈A C D〉 (4), and the earlier redundant path 〈A B D〉 (6) is removed from Reached, but not from Frontier. Why not Frontier, too?

Douglas H. Fisher

Lowest-Cost First Search (aka Uniform Cost Search, aka Dijkstra's Algorithm) of a Graph (With Checking for Repeated Vertices)



Frontier (priority queue of paths)

1. [**⟨A⟩ (0)**]
2. [**⟨A B⟩ (1) ⟨A C⟩ (3)**]
3. [**⟨A B F⟩ (2)** ⟨A C⟩ (3) **⟨A B D⟩ (6)**]
4. [⟨A C⟩ (3) ⟨A B D⟩ (6) **⟨A B F G⟩ (6) ⟨A B F H⟩ (9)**]
5. [**⟨A C D⟩ (4) ⟨A C E⟩ (5)** ⟨A B D⟩ (6) ⟨A B F G⟩ (6) ⟨A B F H⟩ (9)]
6. [⟨A C E⟩ (5) ⟨A B D⟩ (6) ⟨A B F G⟩ (6) ⟨A B F H⟩ (9)]
7. [⟨A B D⟩ (6) ⟨A B F G⟩ (6) **⟨A C E H⟩ (6)** ⟨A B F H⟩ (9)]

Reached

1. { **⟨A⟩ (0)**}
2. { ⟨A⟩ (0) **⟨A B⟩ (1) ⟨A C⟩ (3)**}
3. { ⟨A⟩ (0) ⟨A B⟩ (1) ⟨A C⟩ (3) **⟨A B F⟩ (2) ⟨A B D⟩ (6)**}
4. { ⟨A⟩ (0) ⟨A B⟩ (1) ⟨A C⟩ (3) ⟨A B F⟩ (2) ⟨A B D⟩ (6) **⟨A B F G⟩ (6) ⟨A B F H⟩ (9)**}
5. { ⟨A⟩ (0) ⟨A B⟩ (1) ⟨A C⟩ (3) ⟨A B F⟩ (2) ⟨A B F G⟩ (6) ⟨A B F H⟩ (9) **⟨A C D⟩ (4) ⟨A C E⟩ (5)**}
6. { ⟨A⟩ (0) ... ⟨A B F H⟩ (9) ⟨A C D⟩ (4) ⟨A C E⟩ (5)}
7. { ⟨A⟩ (0) ... ~~⟨A B F H⟩ (9)~~ ⟨A C D⟩ (4) ⟨A C E⟩ (5) **⟨A C E H⟩ (6)**}

A new shorter path to H is discovered, thereby causing an update to Reached. ⟨A C E H⟩ (6) will become part of the final solution.

Douglas H. Fisher

Will Early Goal Test Work for Least-Cost First Search?

No, *not while guaranteeing a least cost solution in any case*!

Look at steps 12–15 of example of previous slide, repeated here:

12. [⟨A C E H K⟩ (8) ⟨A B F H⟩ (9) ⟨A C E H J⟩ (9)]
13. [⟨A B F H⟩ (9) ⟨A C E H J⟩ (9) **⟨A C E H K M⟩ (12)**]
14. [⟨A C E H J⟩ (9) ⟨A C E H K M⟩ (12)]
15. [**⟨A C E H J M⟩ (10)** ⟨A C E H K M⟩ (12)]

If ⟨A C E H K M⟩ (12) were returned immediately after it was found in step 13, and before placing it on Frontier, then ⟨A C E H J M⟩ (10) would not have been discovered in step 15

Embedding Path Information in State Descriptions

Exploring Alternatives With Search

Douglas H. Fisher

A Revision to Generic Search Algorithm for Explicit Graphs

Some observations

1. In the last example of least-cost first search we have this entry in the Frontier:

   10. [〈A B F G I〉 (8) 〈A B F G L〉 (8) **〈A C E H K〉 (8)** 〈A B F H〉 (9) **〈A C E H J〉 (9)**]

   The sub-path 〈A B F〉 is stored thrice, 〈A B F G〉 is stored twice, and 〈A C E H〉 is stored twice.

2. Generally, in both the Frontier and Reached structures, there are redundancies across paths.

3. We can eliminate redundancy while retaining the capability of remembering paths (and returning paths to goals) by distinguishing the vertex and arc space (i.e., the state space) and the search space of that contains information to efficiently recover requisite information such as vertices, arcs, paths, and costs (e.g., an implementation of [〈A B F G I〉 (8)]).

structure **SearchNode**

State (e.g., vertex v) in state space

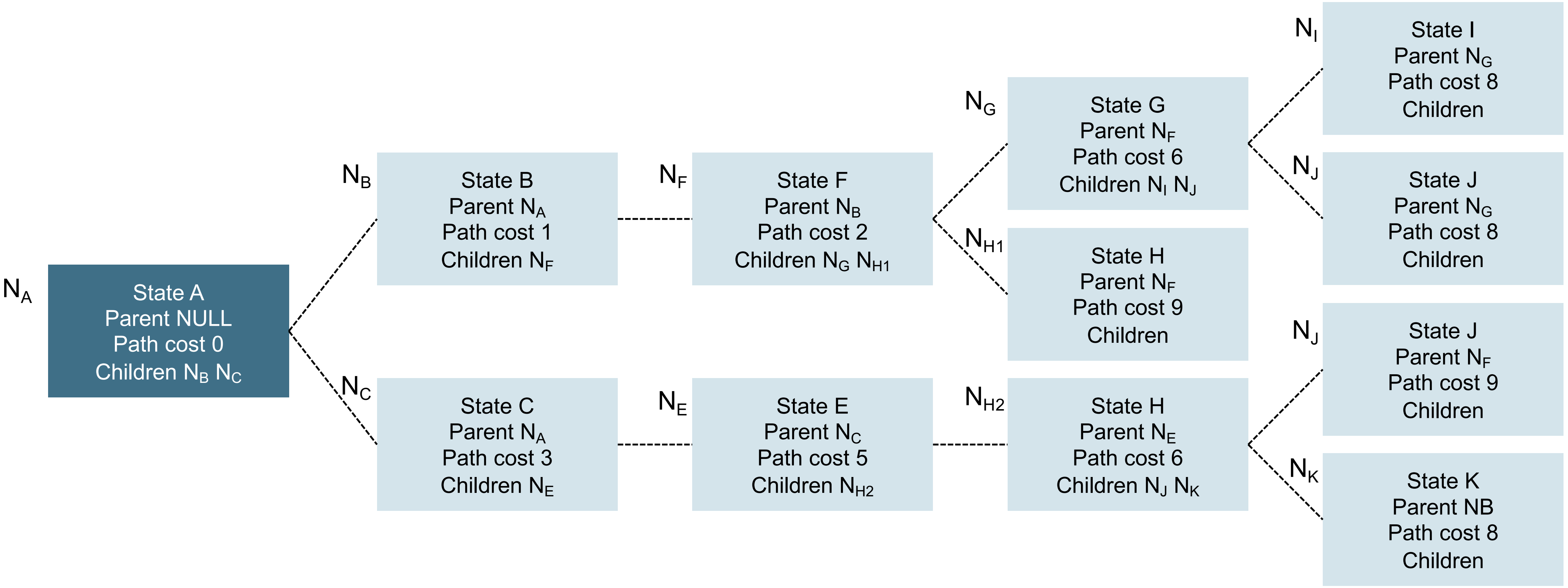
Parent is a SearchNode with state v_k, where v_k is a directed neighbor of v in state space (v_k, v) (accessible by pointer or hashing)

Path-Cost is the cost of the arc (v_k, v) in state space plus the Path Cost of Parent

Children is a set of SearchNodes, each of which corresponds to a reachable neighbor, v’, of v in state space; not every v’ need have an associated child

A Revision to Generic Search Algorithm for Explicit Graphs

structure **SearchNode** (**State Parent Path-Cost Children**)



10.[〈A B F G I〉 (8) 〈A B F G L〉 (8) **〈A C E H K〉 (8)** 〈A B F H〉 (9) **〈A C E H J〉 (9)]**

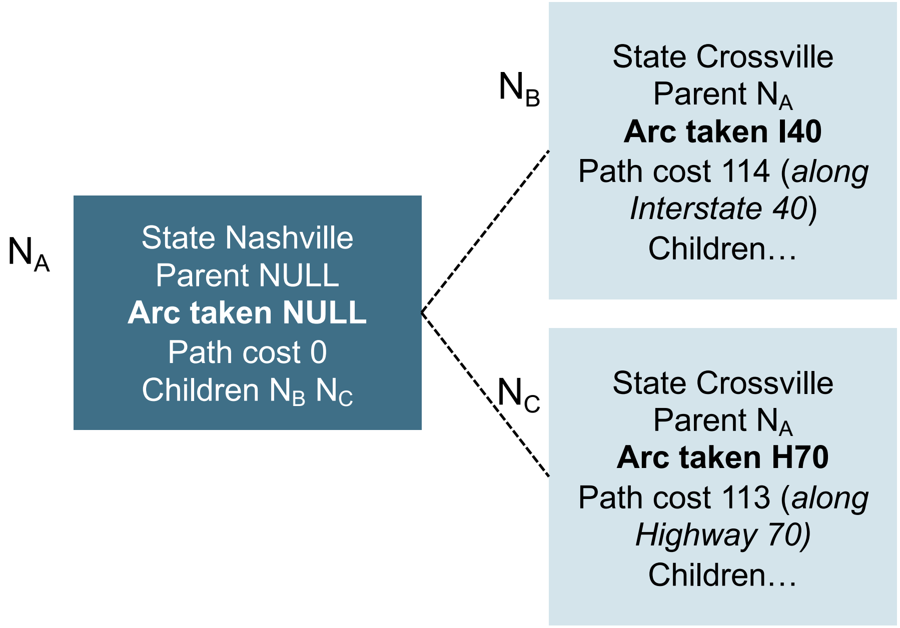
Douglas H. Fisher

A Revision to Generic Search Algorithm for Explicit Graphs

```
structure SearchNode (State Parent Path-Cost Children)
SearchNode Search (Vertices V, Arcs A, v0, G)
  /* … assume that  each entry in A now includes a cost c (vi, vj, c) where c */
  SearchNode N = new SearchNode(State v0, Parent NULL, Path-Cost 0, Children NULL)
  Frontier = [N]
  Reached = {N}
  while Frontier != [ ] do
     select and remove N from Frontier
     if N.State in G then return N   // from which the path from v0 to N.State can be recovered
     for each v such that (vk, v, c) in A
        SearchNode L = new SearchNode(State v, Parent N, Path-Cost N.Path-Cost + c, Children NULL)
        if !exists Node M in Explored s.t. M.State == v or L.Path-Cost < M.Path-Cost
           N.Children = N.Children + L
           Reached = reached - M  + L. //if M doesn't exist then Reached – M is a no-op
           Frontier = Frontier  + L
  return 〈〉
```

Multiple Arcs Between Vertices

The use of a SearchNode structure also facilitates something else. There can be multiple arcs between the same vertices, perhaps with different costs. For example, a mapping app can consider two different direct routes between two towns, one along highway 70 and one along Interstate 40. In this case, we would probably want to store the arc taken from parent to child with each node as well to disambiguate.

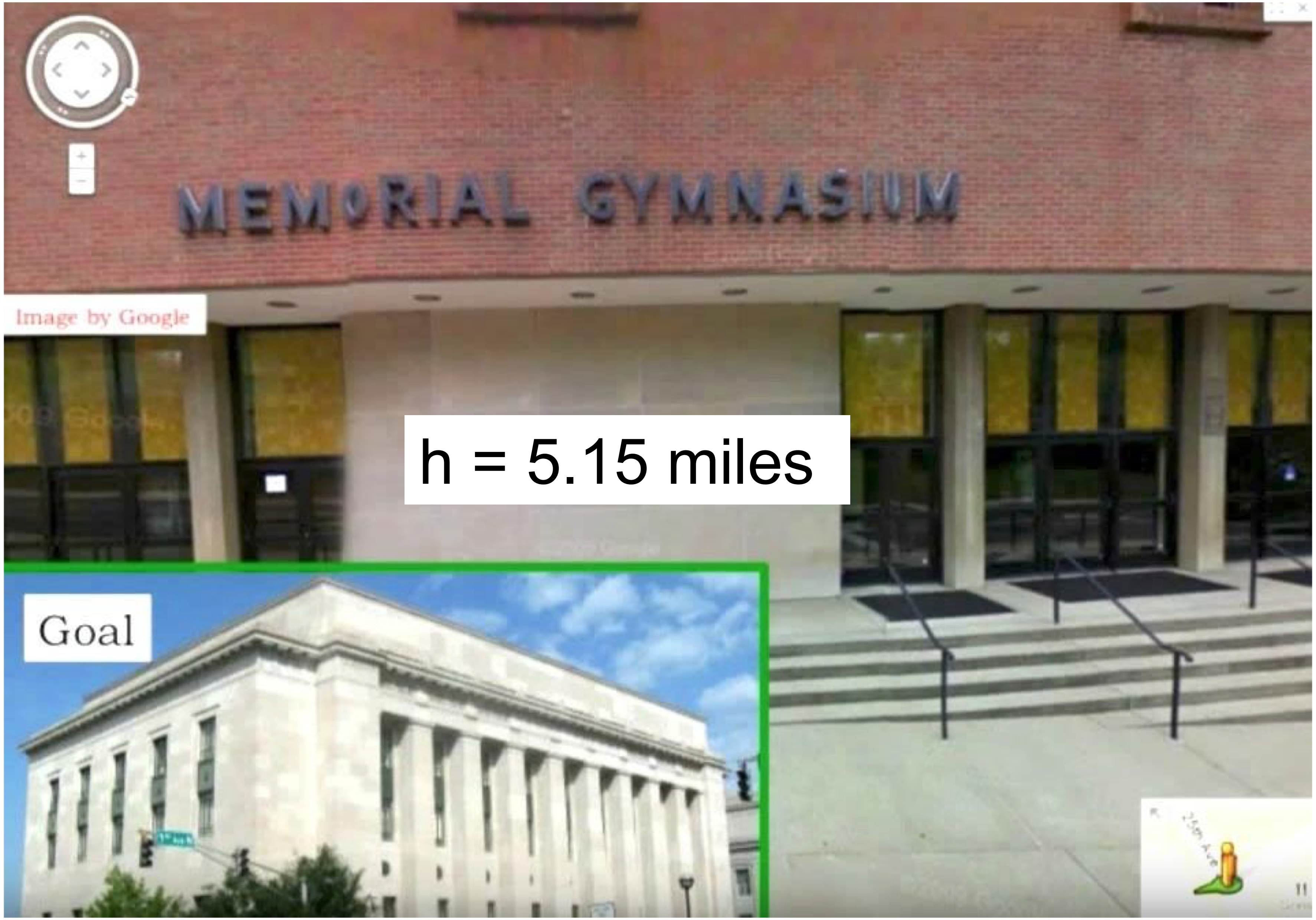


Informed (or Heuristic) Search of an Explicit Graph

Exploring Alternatives With Search

Douglas H. Fisher


Tennessee
Supreme
Court
5 miles
h (Westend via Kessington)
5.3 miles
h (25th Ave Garage by Jess Neely)
Westend via Kessington
Cost (WvK)
Memorial Gym
cost(JNa25)
Jess Neely and 25th Ave Garage


MEMORIAL GYMNASIUM
Image by Google
h = 5.15 miles
Goal

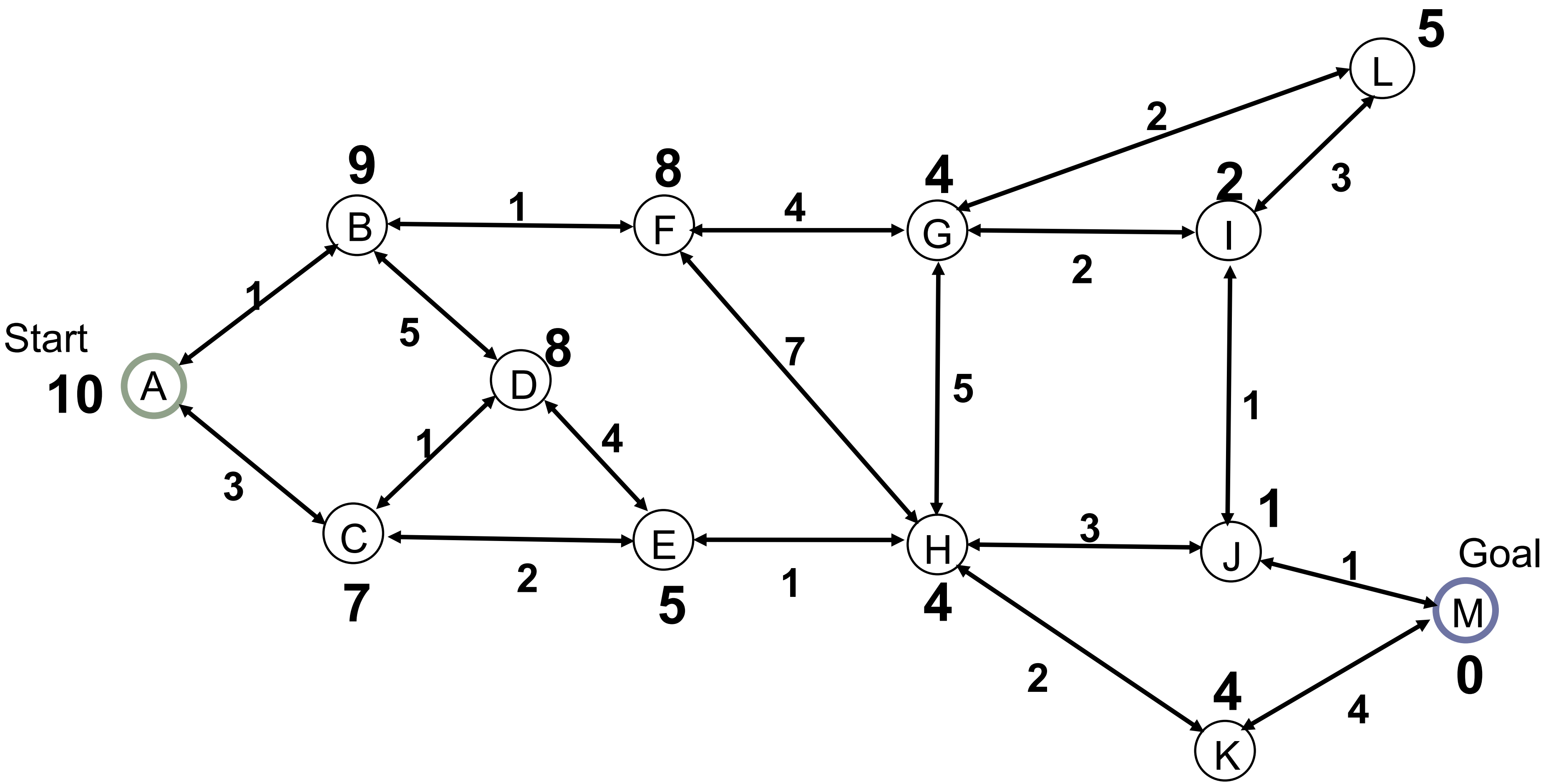

Action: go right
towards
Jess Neely (and
25th Ave Garage)
25th Ave

Douglas H. Fisher

Douglas H. Fisher


5.0 miles
West End Ave
After applying Westend
“operator”

An Example Graph



Douglas H. Fisher

An Example Graph (cont.)

In the graph of the previous slide, which we will use going forward:

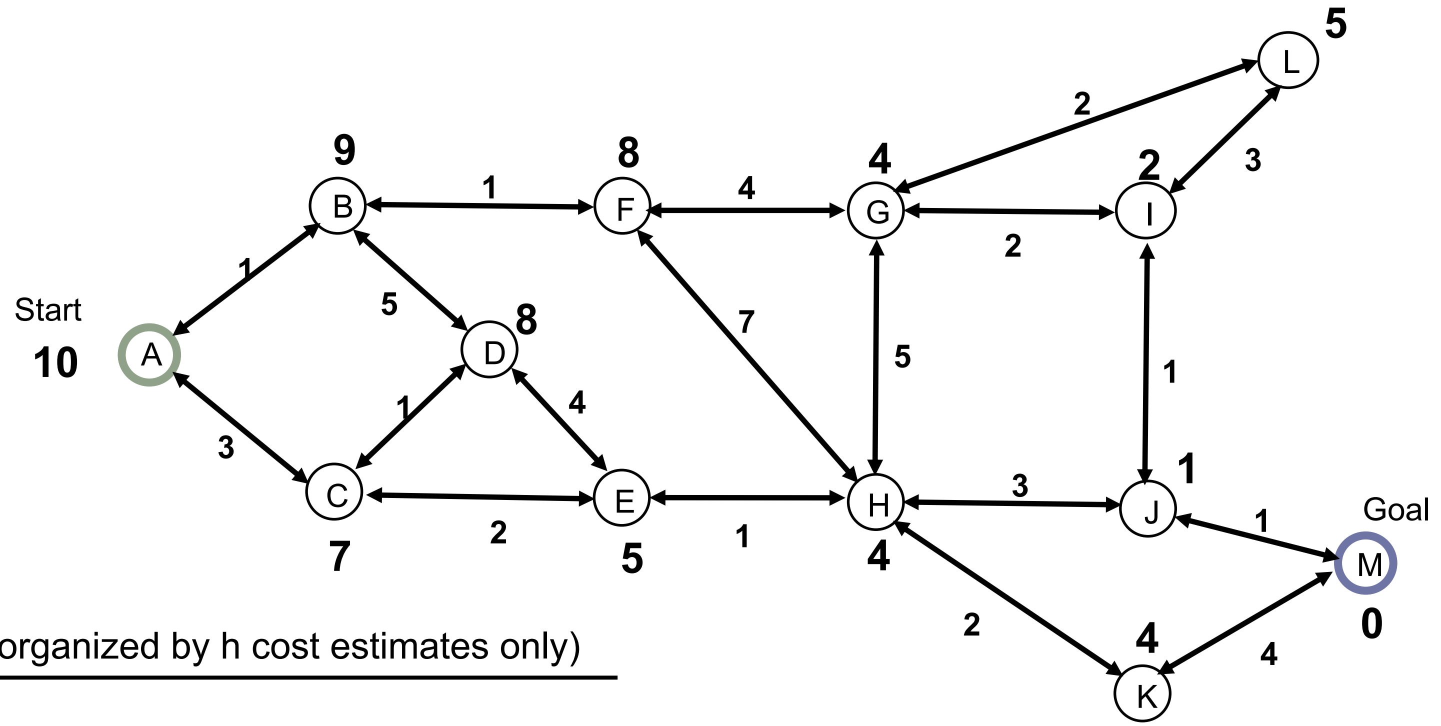
- Arc costs, also called g costs, label arcs, again under the assumption that costs are the same in each direction, which is not necessary or even typical
- Heuristic estimates of remaining cost, called h, from each vertex to a goal (M) along least-cost path label each vertex
- In this example, the h cost of each node happens to be exact; this would be rare, but we'll start with this illustration
- Though we learned a representation for the search space that used a SearchNode structure, which comes with space advantages, we will continue representing paths separately for ease of illustration

Greedy Best-First Search of an Explicit Graph

Exploring Alternatives With Search

Douglas H. Fisher

Greedy Best-First Search



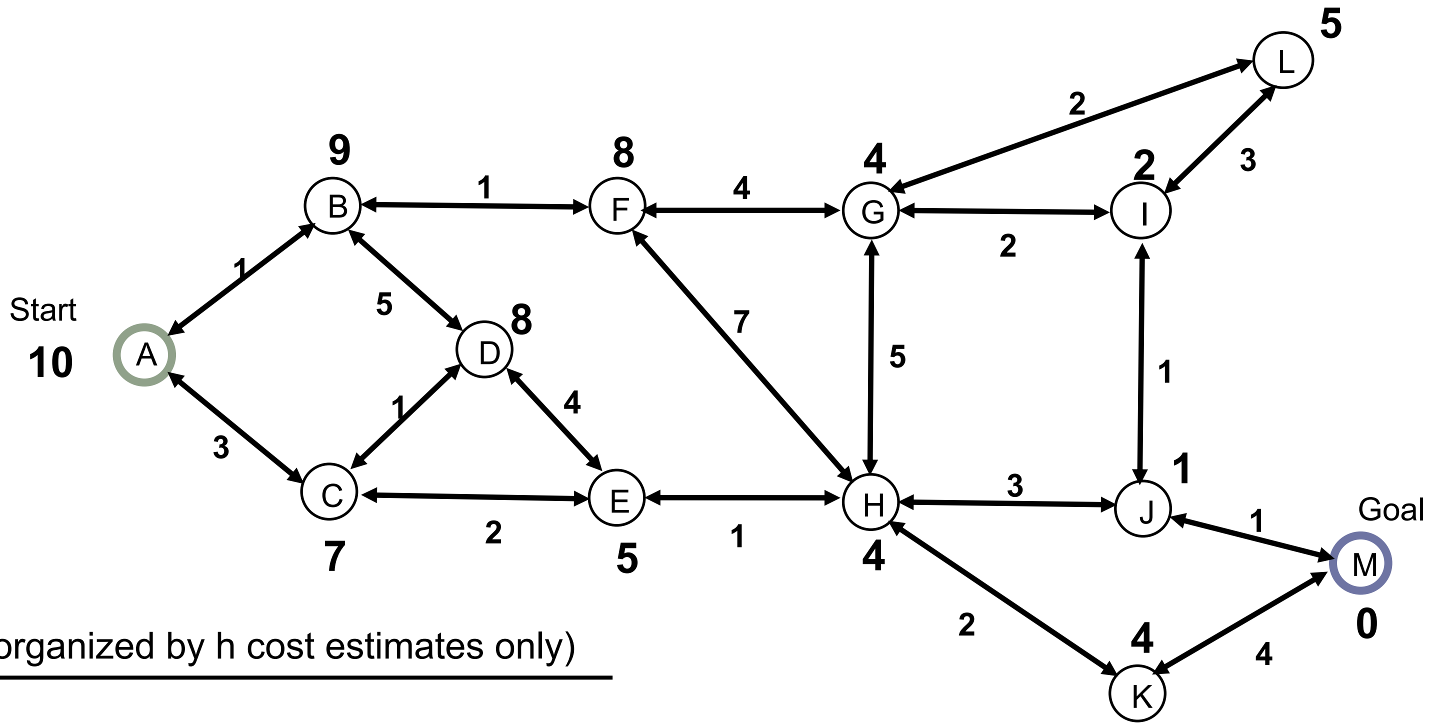
Frontier (priority queue organized by h cost estimates only)

1. **[〈A〉 (10)]**
2. **[〈A C〉 (7) 〈A B〉 (9)]**
3. **[〈A C E〉 (5) 〈A C D〉 (8)** 〈A B〉 (9)]
4. **[〈A C E H〉 (4)** 〈A C D〉 (8) 〈A B〉 (9)]

Reached is not shown, but it is still computed and used to censor **〈A C E D〉 (8)** and **〈A C E C〉 (7)** in step 4 after **〈A C E〉 (5)** is expanded in step 3, for example.

Douglas H. Fisher

Greedy Best-First Search (cont.)



Frontier (priority queue organized by h cost estimates only)

1. **[⟨A⟩ (10)]**
2. **[⟨A C⟩ (7) ⟨A B⟩ (9)]**
3. **[⟨A C E⟩ (5) ⟨A C D⟩ (8)** ⟨A B⟩ (9)]
4. **[⟨A C E H⟩ (4)** ⟨A C D⟩ (8) ⟨A B⟩ (9)]
5. **[⟨A C E H J⟩ (1) ⟨A C E H K⟩ (4) ⟨A C E H G⟩ (4)** ⟨A C D⟩ (8) **⟨A C E H F⟩ (8)** ⟨A B⟩ (9)]
6. [**⟨A C E H J M⟩ (0) ⟨A C E H J I⟩ (2)** ⟨A C E H K⟩ (4) ⟨A C E H G⟩ (4) ⟨A C D⟩ (8) ⟨A C E H F⟩ (8) ⟨A B⟩ (9)]

Douglas H. Fisher

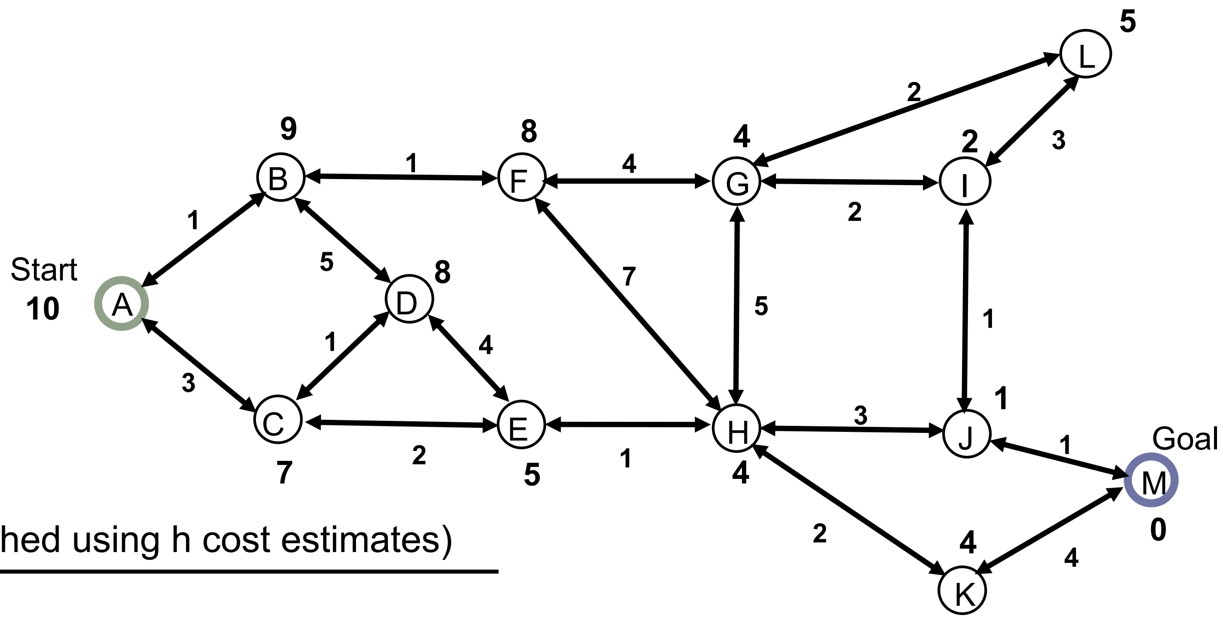
Heuristic Depth-First Search of an Explicit Graph

Exploring Alternatives With Search

Douglas H. Fisher

Heuristic Depth-First Search

Regular DFS, but on each expansion, push children in inverse order by h (highest to lowest)



Frontier (stack with siblings pushed using h cost estimates)

1. **[⟨A⟩ (10)]**
2. **[⟨A C⟩ (7) ⟨A B⟩ (9)]**
3. **[⟨A C E⟩ (5) ⟨A C D⟩ (8)** ⟨A B⟩ (9)]
4. **[⟨A C E H⟩ (4)** ⟨A C D⟩ (8) ⟨A B⟩ (9)]
5. **[⟨A C E H J⟩ (1) ⟨A C E H K⟩ (4) ⟨A C E H G⟩ (4) *⟨A C E H F⟩ (8)*** ⟨A C D⟩ (8) ⟨A B⟩ (9)]
6. [**⟨A C E H J M⟩ (0) ⟨A C E H J I⟩ (2)** ⟨A C E H K⟩ (4) ⟨A C E H G⟩ (4) ⟨A C E H F⟩ (8) ⟨A C D⟩ (8) ⟨A B⟩ (9)]

Almost no difference between heuristic depth-first search and greedy best-first search in this example, just in placement of ***⟨A C E H F⟩ (8)***

Douglas H. Fisher

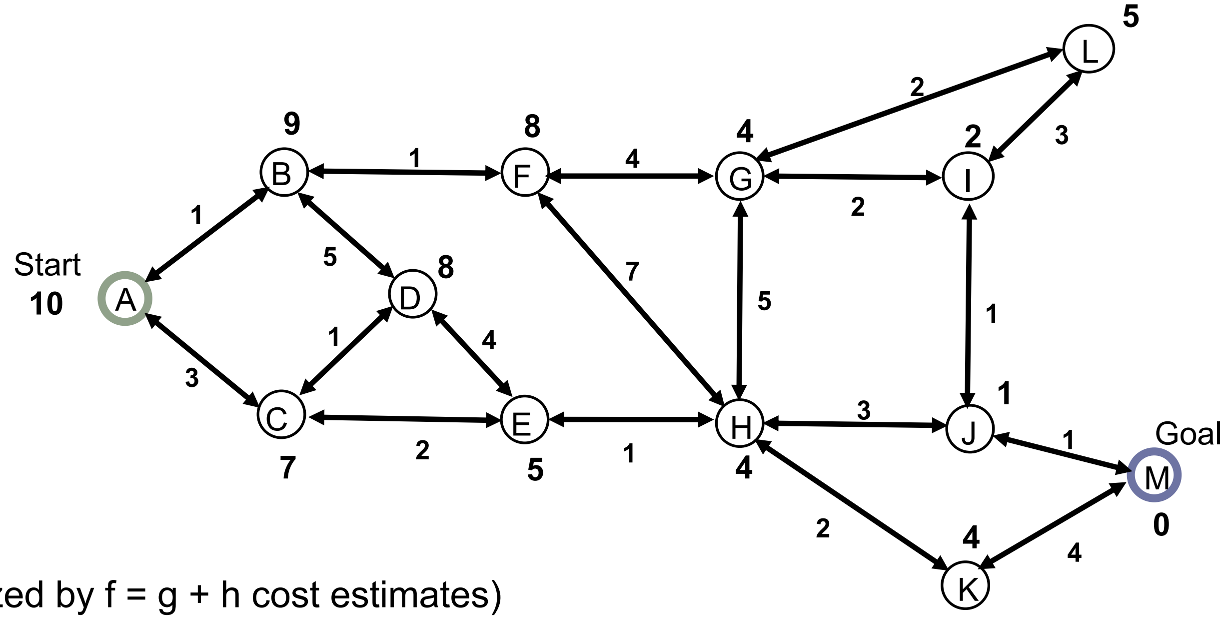
A* Search of an Explicit Graph

Exploring Alternatives With Search

Douglas H. Fisher

A*

Use both actual cost so far plus (g) estimated cost to go (h). This sum is called f.

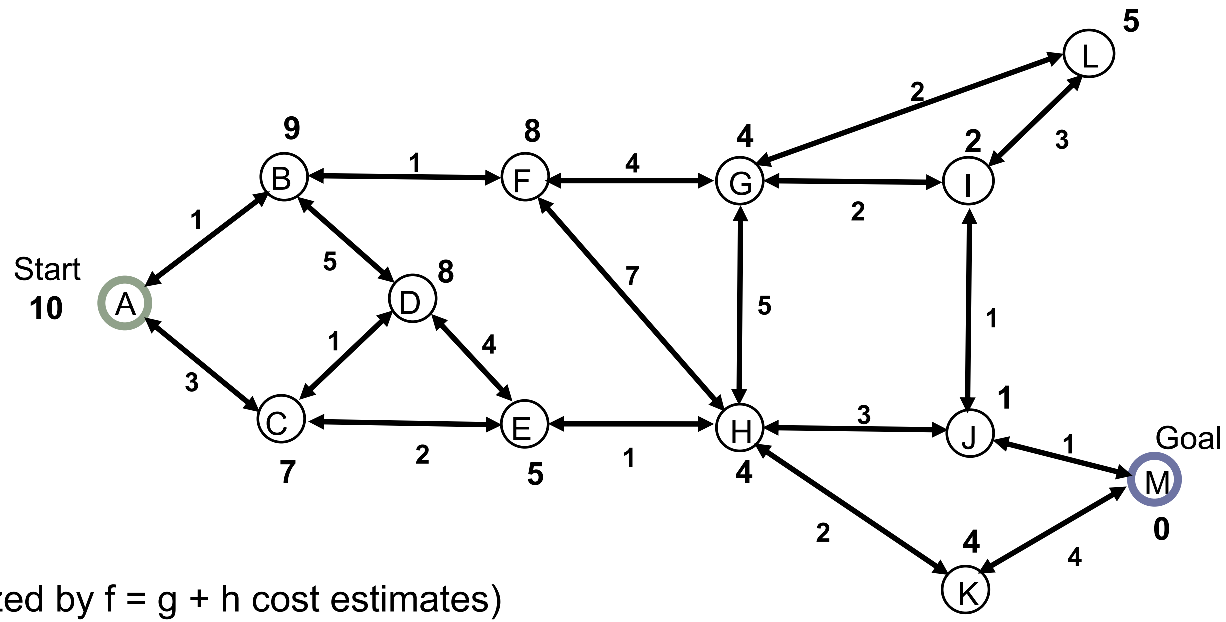


Frontier (priority queue organized by f = g + h cost estimates)

1. **[〈A〉 (10)]**
2. **[〈A B〉 (10) 〈A C〉 (10)]**
3. **[〈A B F〉 (10)** 〈A C〉 (10) **〈A B D〉 (14)]**
4. **[〈A B F G〉 (10)** 〈A C〉 (10) **〈A B F H〉 (13)** 〈A B D〉 (14)]

- Reached is not shown, but it is still being used to prevent redundant paths.
- Note that, in cases of ties, the most recent generated path is placed first. This is unlike previous examples. Might there be (dis)advantages to this practice?

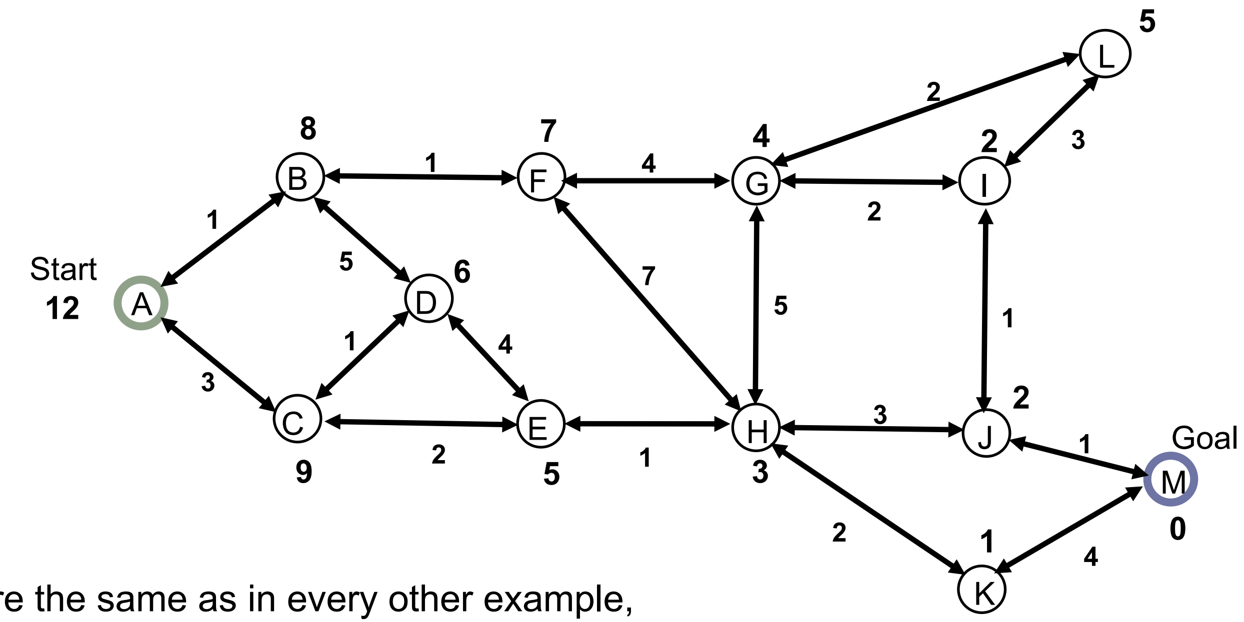
A* (cont.)



Frontier (priority queue organized by f = g + h cost estimates)

1. **[〈A〉 (10)]**
2. **[〈A B〉 (10) 〈A C〉 (10)]**
3. **[〈A B F〉 (10)** 〈A C〉 (10) **〈A B D〉 (14)]**
4. **[〈A B F G〉 (10)** 〈A C〉 (10) **〈A B F H〉 (13)** 〈A B D〉 (14)]
5. **[〈A B F G I〉 (10)** 〈A C〉 (10) **〈A B F G L〉 (13)** 〈A B F H〉 (13) 〈A B D〉 (14)]
6. **[〈A B F G I J〉 (10)** 〈A C〉 (10) 〈A B F G L〉 (13) 〈A B F H〉 (13) 〈A B D〉 (14)]
7. **[〈A B F G I J M〉 (10)** 〈A C〉 (10) 〈A B F G L〉 (13) 〈A B F H〉 (13) 〈A B D〉 (14)

Second Example Graph



- In this example, g costs are the same as in every other example, but the h costs have changed.
- Is the heuristic admissible?
- Perform greedy best-first, heuristic depth-first, and A* search on this graph.

Iterative Deepening

Exploring Alternatives With Search

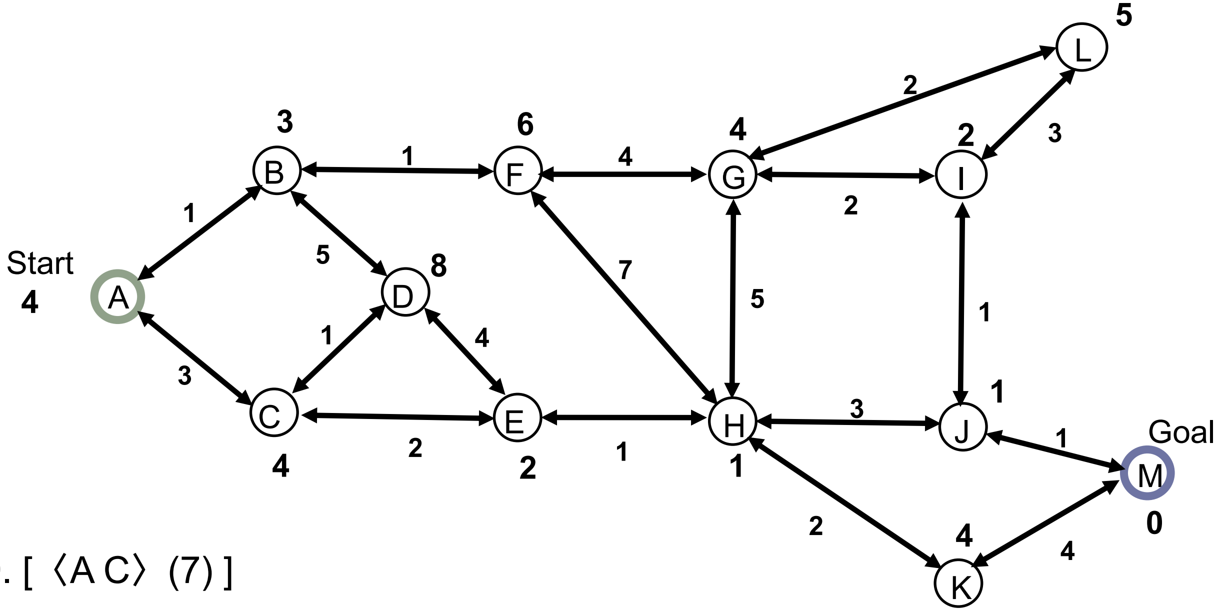
Douglas H. Fisher

Iterative Deepening A*

Keep searching to increasing f-thresholds until a goal is found. The first goal found is guaranteed to be a least cost from start to goal IF h is admissible.

Frontier (Stack)

1. **[⟨A⟩ (4)] //DFS to f-threshold of 4**
2. **[⟨A B⟩ (4) ⟨A C⟩ (7)]**
3. **[⟨A B F⟩ (8) ⟨A B D⟩ (14) ⟨A B A⟩ (6)** ⟨A C⟩ (7) **]**
4. [⟨A B D⟩ (14) ⟨A B A⟩ (6) ⟨A C⟩ (7)]
5. [⟨A B A⟩ (6) ⟨A C⟩ (7)]
6. [⟨A C⟩ (7)] (followed by [ ])
7. **[⟨A⟩ (4)] //DFS to f-threshold of 6**
8. **[⟨A B⟩ (4) ⟨A C⟩ (7)]**
9. **. . .**
10. **[⟨A⟩ (4)] //DFS to f-threshold of 7**
11. **[⟨A B⟩ (4) ⟨A C⟩ (7)]**
12. **[⟨A B F⟩ (8) ⟨A B D⟩ (14) ⟨A B A⟩ (6)** ⟨A C⟩ (7) **]**
13. [⟨A B D⟩ (14) ⟨A B A⟩ (6) ⟨A C⟩ (7)]
14. [**⟨A B A B⟩ (6) ⟨A B A C⟩ (8)** ⟨A C⟩ (7)]
15. [**⟨ A B A B A⟩ (8) ⟨ A B A B F⟩ (10) ⟨ A B A B D⟩ (16)** ⟨A C⟩ (7)] . . .



19. [⟨A C⟩ (7)]
20. [**⟨A C D⟩ (12) ⟨A C E⟩ (7) ⟨A C A⟩ (10)**]
21. [⟨A C D⟩ (12) ⟨A C E⟩ (7) ⟨A C A⟩ (10)]
22. [⟨A C E⟩ (7) ⟨A C A⟩ (10)]
23. **[⟨A C E H⟩ (7) ⟨A C E D⟩ (17) ⟨A C E C⟩ (11)** ⟨A C A⟩ (10)]
24. **[⟨A C E H F⟩ (19) …** ⟨A C A⟩ (10)]

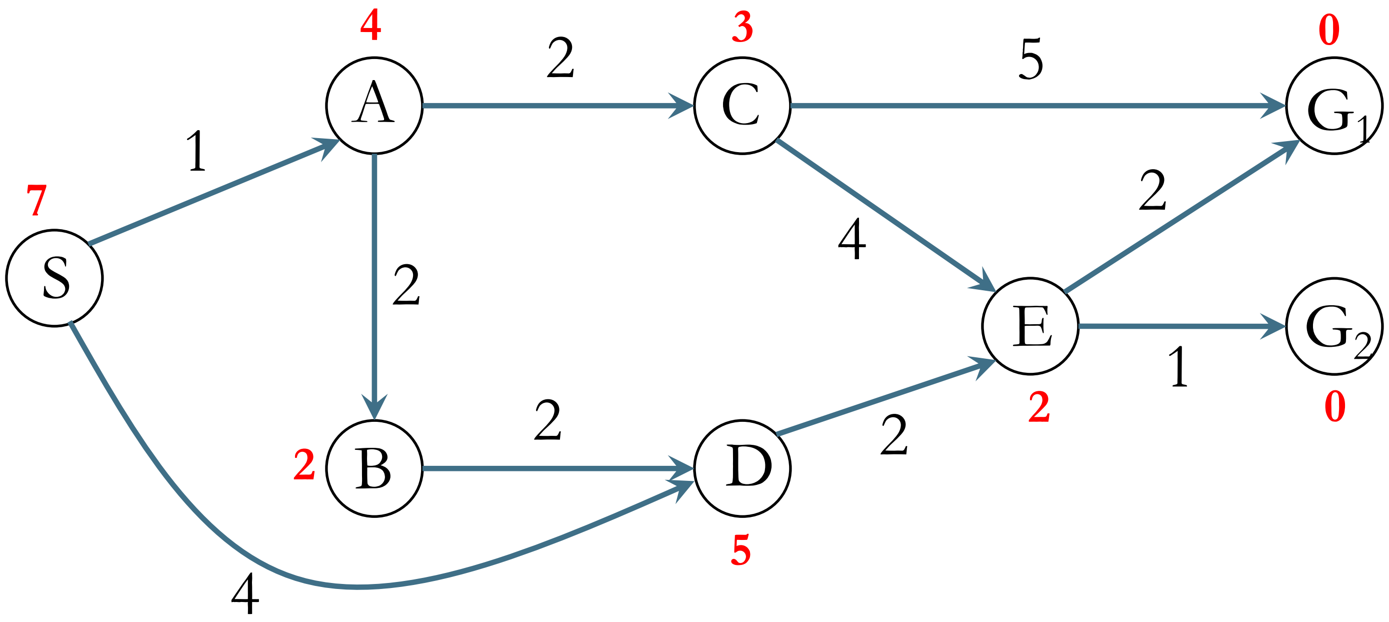
…

30. **[⟨A⟩ (4)] //DFS to f-threshold of 8**

Douglas H. Fisher

Suggested Exercises

Consider the search graph below. The **h value of a node** is given adjacent to that node. The actual cost of traversing an arc (in the indicated directions) is given adjacent to that arc. Node S is the start/initial state. Nodes G_1 and G_2 are goals. Use this graph for the questions to follow.



When you have completed all the questions, upload a pdf of the questions and answers to Brightspace. You may consult the pdf while you take the "quiz" component.

Douglas H. Fisher

Suggested Exercises

1. Give the order in which nodes are visited (i.e., checked for goalness) by **heuristic depth first search**. In the case of two or more nodes with the same evaluation score on the frontier, break the tie by visiting the nodes in alphabetical order as labeled above – this same convention applies to the remaining parts of this question. For this question ONLY, assume that "reached" (as described in the videos) is NOT used.

2. Give the order in which nodes are visited (i.e., checked for goalness) by **greedy best-first search**.

3. Give the order in which nodes are visited (i.e., checked for goalness) by **lowest cost first search**.

4. Give the order in which nodes are visited (i.e., checked for goalness) by **A***.

5. Which nodes would be checked for goalness on the first iteration of **iterative deepening A***?

Notes:

- G1 is alphabetically before G2
- A misconception on the part of some is that the "order that nodes are visited" is the same as "the final path returned". This is not typically the case. Most search strategies will visit vertices that are not part of the final path.

Searching an Implicit Graph

Exploring Alternatives With Search

Douglas H. Fisher

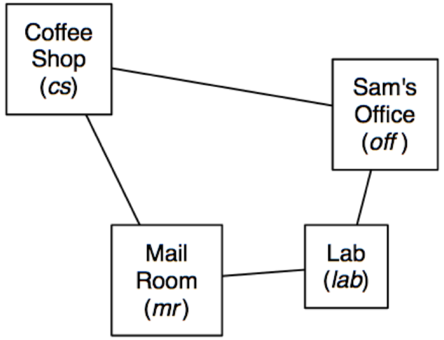
Searching an Implicit Graph

- All of the search methods studied for an explicit graph can be adapted straightforwardly to search of an implicit graph.
- An implicit graph is one with "vertices" (states) that are created "on demand," as search proceeds.
- As with explicit graphs, we are generally most interested in using search to find one or more paths to a goal, rather than simply finding a goal per se.
- Thus, search is used to find a "plan" in virtual space that can be executed in the real world later.

Searching an Implicit Graph (cont.)

Delivery Robot Example

This may look like another problem of searching a graph for a location (vertex) that satisfies some goal condition, but it's **not**!



Features:
RLoc – Rob's location
RHC – Rob has coffee
SWC – Sam wants coffee
MW – Mail is waiting
RHM – Rob has mail

Actions:
mc – move clockwise
mcc – move counterclockwise
puc – pickup coffee
dc – deliver coffee
pum – pickup mail
dm – deliver mail

Rather, the task for service robot Rob is to find a path of actions from a given situation defined by features on the left (e.g., human Sam wants coffee but has none), for a goal situation (e.g., that Sam has coffee).

"Operation" is synonymous with "action."

Douglas H. Fisher

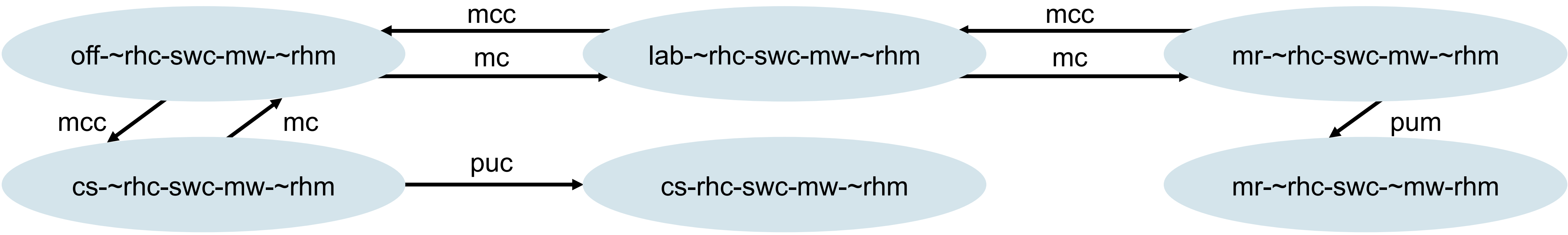
Delivery Robot Example

We could treat this problem like an explicit graph problem, with each situation description as an atomic, indivisible vertex, and vertices are connected by labeled, directed arcs. I'll write each vertex as lab-rhc-swc-mw-rhm (with hyphens) to stress the indivisibility.

If every situation is really indivisible, then lab-rhc-swc-mw-rhm has no more in common with off-rhc-swc-mw-rhm than it does with off-~rhc-~swc-~mw-~rhm.

Source vertex	Arc	Resulting vertex
lab-rhc-swc-mw-rhm	mc	mr-rhc-swc-mw-rhm
lab-rhc-swc-mw-~rhm	mc	mr-rhc-swc-mw-~rhm
lab-rhc-swc-~mw-rhm	mc	mr-rhc-swc-~mw-rhm
lab-rhc-swc-~mw-~rhm	mc	mr-rhc-swc-~mw-~rhm
lab-rhc-~swc-mw-rhm	mc	mr-rhc-~swc-mw-rhm
lab-rhc-~swc-mw-~rhm	mc	mr-rhc-~swc-mw-~rhm
lab-rhc-~swc-~mw-rhm	mc	mr-rhc-~swc-~mw-rhm
lab-rhc-~swc-~mw-~rhm	mc	mr-rhc-~swc-~mw-~rhm
lab-~rhc-swc-mw-rhm	mc	mr-~rhc-swc-mw-rhm
…		
lab-~rhc-~swc-~mw-~rhm	mc	mr-~rhc-~swc-~mw-~rhm
…		

2^4 = 16 different vertex-arc-vertex triples are needed to represent that the move-clockwise (mc) action will move Rob from the lab to the mail room (mr). And that's only the start!

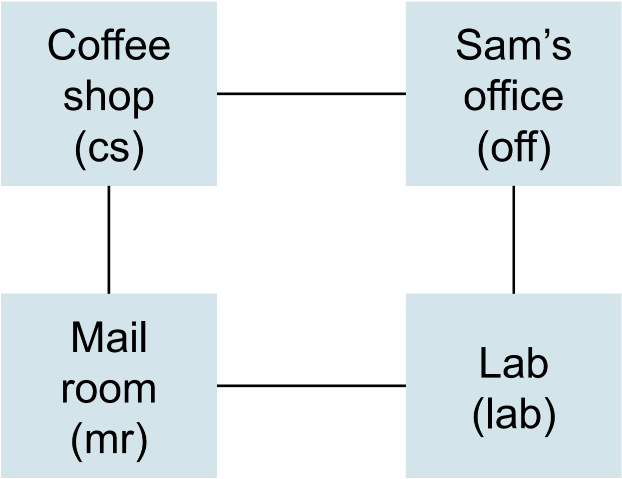


Douglas H. Fisher

Delivery Robot Example

A factored (or feature vector or attribute-value pairs) representation

- rloc (Rob’s location) is four-valued.
- rhc (Rob has coffee) is binary-valued.
- swc (Sam wants coffee) is binary-valued.
- mw (mail waiting) is binary-valued.
- rhm (Rob has mail) is binary-valued.



State	Action	Resulting State
< lab, rhc, swc, mw, rhm>	mc	< mr, rhc, swc, mw, rhm>
< lab, rhc, swc, mw, ~rhm>	mc	< mr, rhc, swc, mw, ~rhm>
< lab, rhc, swc, ~mw, rhm>	mc	< mr, rhc, swc, ~mw, rhm>
< lab, rhc, swc, ~mw, ~rhm>	mc	< mr, rhc, swc, ~mw, ~rhm>
…		
< lab, ~rhc, ~swc, ~mw, ~rhm>	mc	< mr, ~rhc, ~swc, ~mw, ~rhm>

16 mc

<lab, ?V1, ?V2, ?V3, ?V4>	**mc**	**<mr, ?V1, ?V2, ?V3, ?V4>**
<mr, ?V1, ?V2, ?V3, ?V4>	**mc**	**<cs, ?V1, ?V2, ?V3, ?V4>**
<cs, ?V1, ?V2, ?V3, ?V4>	**mc**	**<off, ?V1, ?V2, ?V3, ?V4>**
<off, ?V1, ?V2, ?V3, ?V4>	**mc**	**<lab, ?V1, ?V2, ?V3, ?V4>**

Represented as four patterns with factored representation

Different representations for actions possible, (e.g., perhaps Rob can’t be holding coffee to pick up mail) but must choose one set of definitions.

<cs, ~rhc, ?V1, ?V2, ?V3>	**puc**	**<cs, rhc, ?V1, ?V2, ?V3>**
<off, rhc, ?V1, ?V2, ?V3>	**dc**	**<off, ~rhc, ~swc, ?V2, ?V3>**
<mr, ?V1, ?V2, mw, ~rhm>	**pum**	**<mr, ?V1, ?V2, ~mw, rhm>**
<off, ?V1, ?V2, ?V3, rhm>	**dm**	**<off, ?V1, ?V2, ~V3, ~rhm>**

Delivery Robot Example

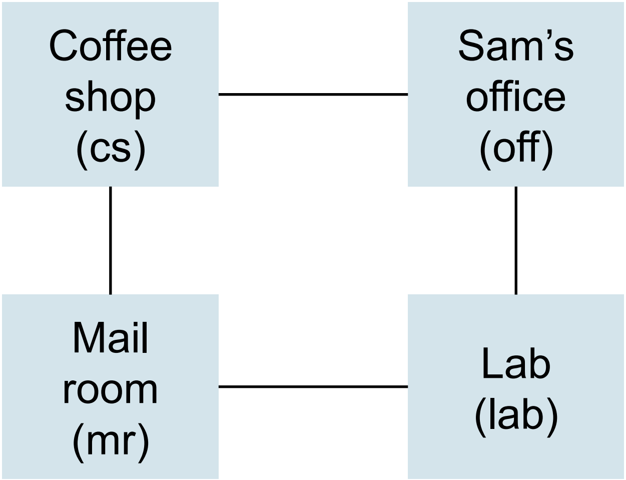
- rloc (Rob's location) is four-valued.
- rhc (Rob has coffee) is binary-valued.
- swc (Sam wants coffee) is binary-valued.
- mw (mail waiting) is binary-valued.
- rhm (Rob has mail) is binary-valued.

Initial state

<cs, ~rhc, swc, mw, ~rhm>

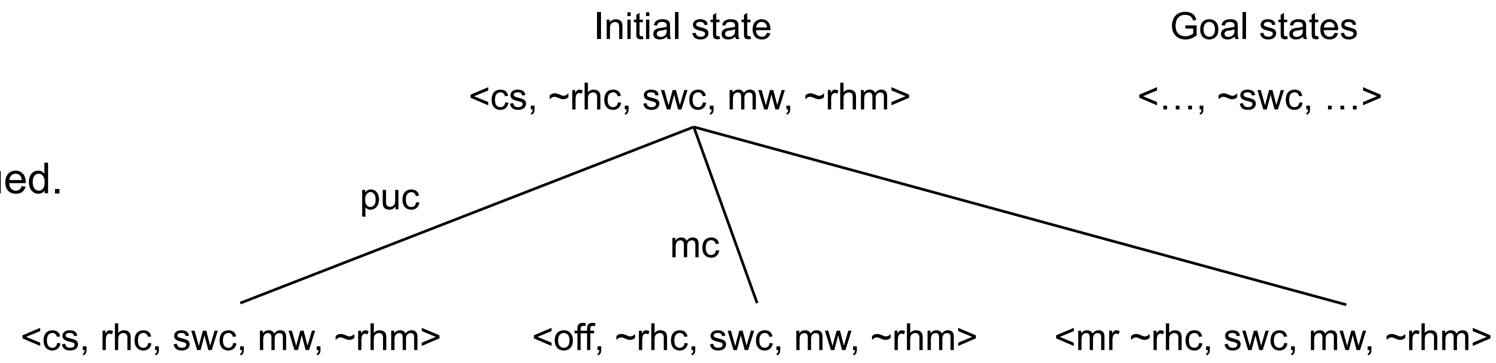
Goal states

<?V1, ?V2, ~swc, ?V3, ~V4>

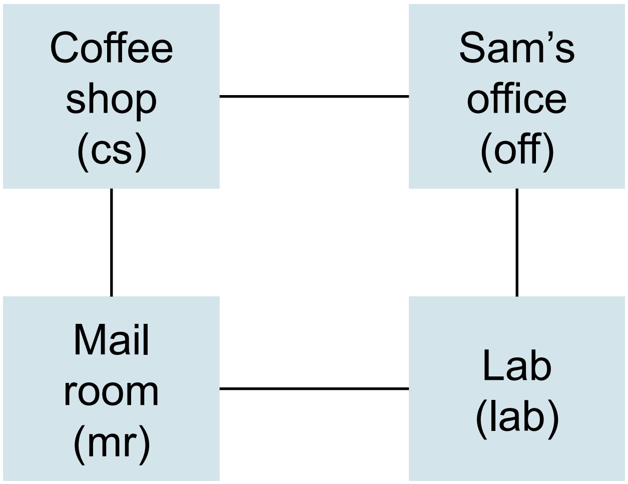


Delivery Robot Example

- rloc (Rob’s location) is four-valued.
- rhc (Rob has coffee) is binary-valued.
- swc (Sam wants coffee) is binary-valued.
- mw (mail waiting) is binary-valued.
- rhm (Rob has mail) is binary-valued.

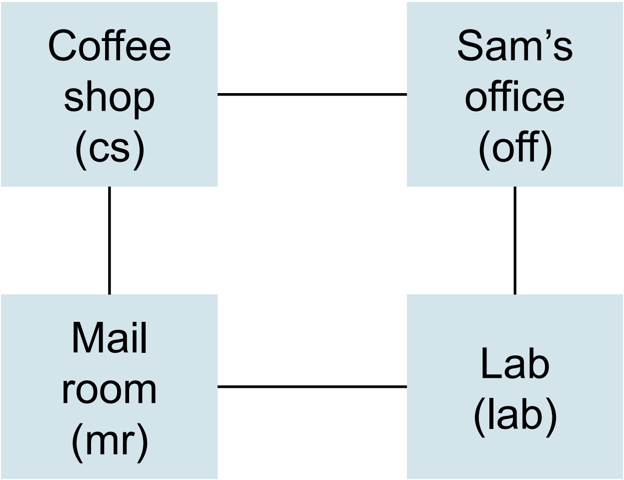


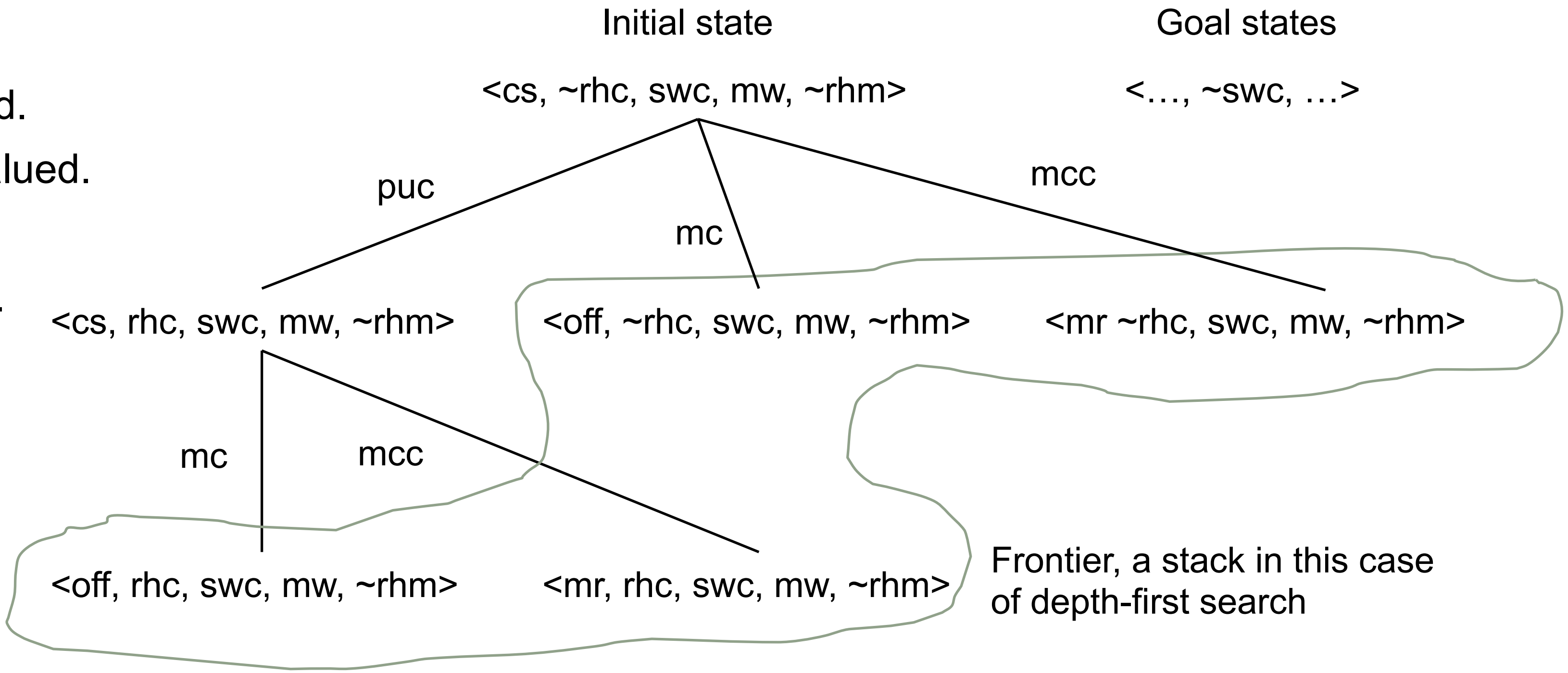
States are realized through operator application.



Delivery Robot Example

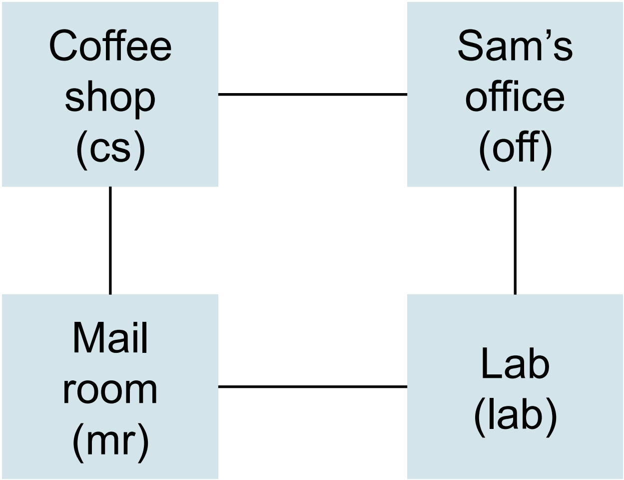
- rloc (Rob's location) is four-valued.
- rhc (Rob has coffee) is binary-valued.
- swc (Sam wants coffee) is binary-valued.
- mw (mail waiting) is binary-valued.
- rhm (Rob has mail) is binary-valued.

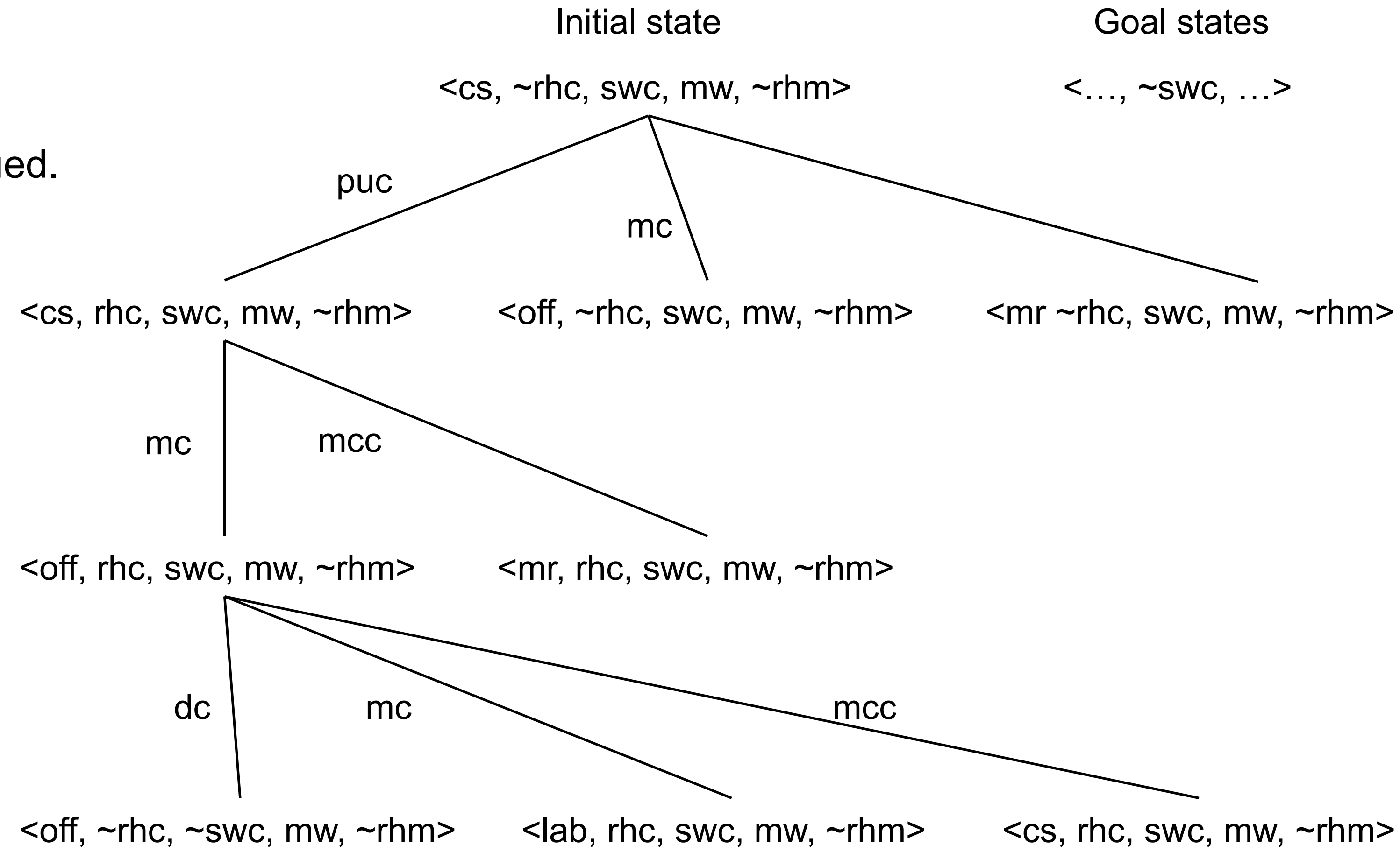




Delivery Robot Example

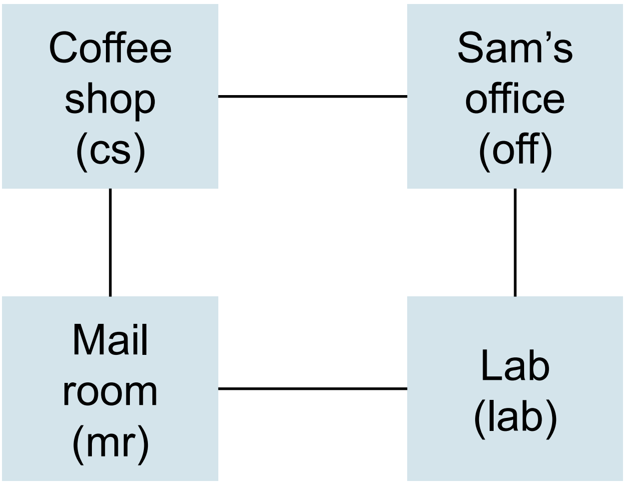
- rloc (Rob’s location) is four-valued.
- rhc (Rob has coffee) is binary-valued.
- swc (Sam wants coffee) is binary-valued.
- mw (mail waiting) is binary-valued.
- rhm (Rob has mail) is binary-valued.

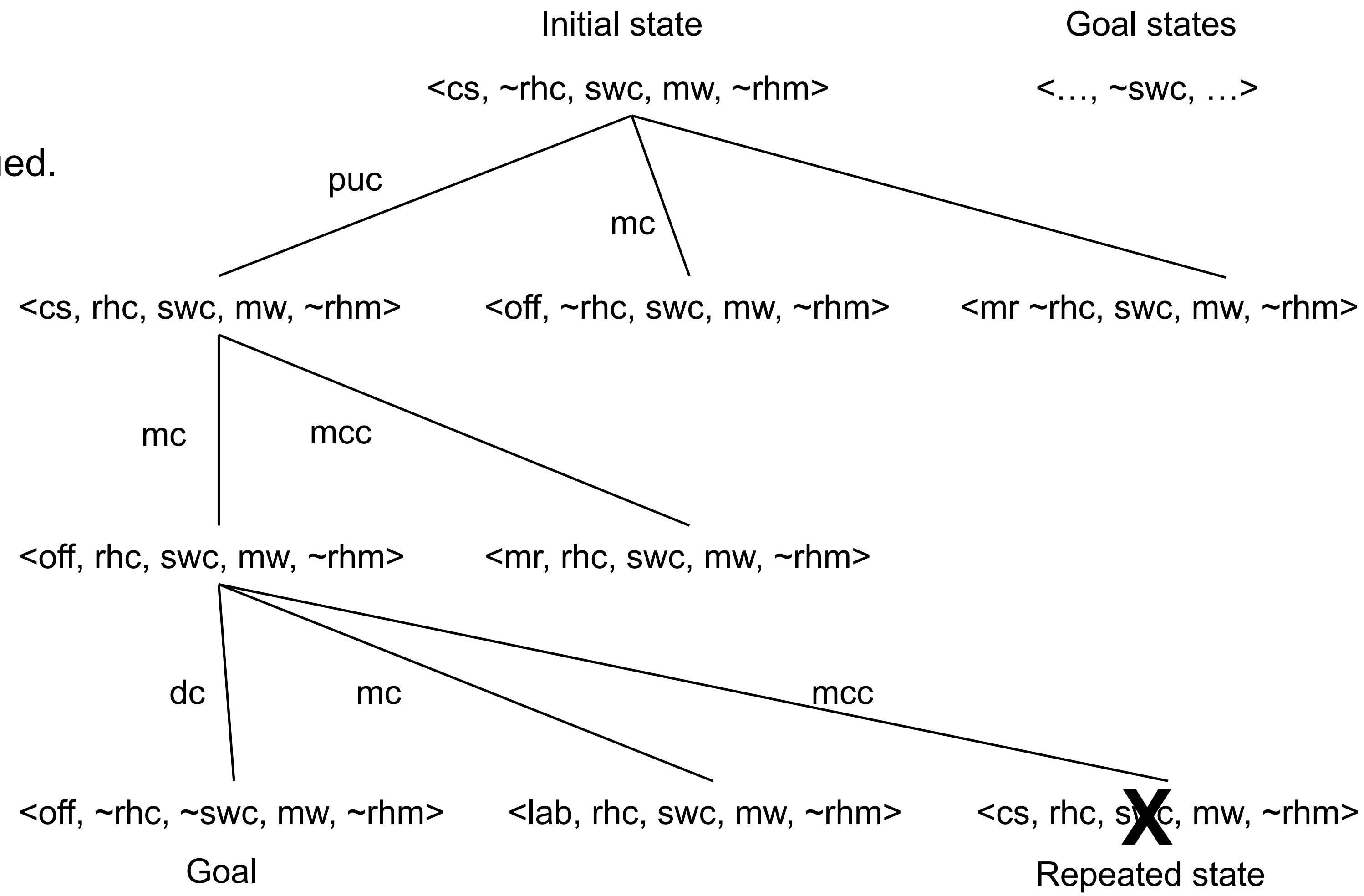




Delivery Robot Example

- rloc (Rob's location) is four-valued.
- rhc (Rob has coffee) is binary-valued.
- swc (Sam wants coffee) is binary-valued.
- mw (mail waiting) is binary-valued.
- rhm (Rob has mail) is binary-valued.





Douglas H. Fisher

Searching an Implicit Graph: A World Trade Game

Exploring Alternatives With Search

Douglas H. Fisher

World Trade Game Example

A simulation in which fictional countries that are actually fronts for AI game players build and trade resources in pursuit of bettering each of their own circumstances as well as the fictional world's circumstances, each country using utility metrics of their software designer's and AI's choosing.

Country	Population (M)	Metalic Elements (IU)	Timber (IU)	Metallic Alloys (IU)	Metallic Alloys Waste (IU)	Electronics (IU)	Electronics Waste (IU)	Housing (M)	Housing Waste (IU)
Atlantis	p_A	n_{AME}	n_{AT}	n_{AMA}	n_{AMA}	n_{AE}	n_{AEW}	n_{AH}	n_{AHW}
Brobding-nag	p_B	n_{BME}	n_{BT}	n_{BMA}	n_{BMA}	n_{BE}	n_{BEW}	n_{BH}	n_{BHW}
Carpania	p_C	n_{CME}	n_{CT}	n_{CMA}	n_{CMA}	n_{CE}	n_{CEW}	n_{CH}	n_{CHW}
Dinotopia	p_D	n_{DME}	n_{DT}	n_{DMA}	n_{DMA}	n_{DE}	n_{DEW}	n_{DH}	n_{DHW}
Erewhon	p_E	n_{EME}	n_{ET}	n_{EMA}	n_{EMA}	n_{EE}	n_{EEW}	n_{EH}	n_{EHW}

World Trade Game Example

TRANSFORMs are within-country actions that allow a country, given by the value of variable ?C, to create composite resources (OUTPUTS) from raw resources and other composite resources (INPUTS). Templates show relative amounts of resources, which can be multiplicatively adjusted, so that the Alloys Template can be used to transform INPUTS of 3*1 population and 3*2 MetallicElements into OUTPUTS of 3*1 Population, 3*1 MetallicAlloys, and 3*MettalicAlloysWaste.

Housing template	Alloys template	Electronics template
(TRANSFORM ?C (INPUTS (Population 5) (MetallicElements 1) (Timber 5) (MetallicAlloys 3)) (OUTPUTS (Housing 1) (HousingWaste 1) (Population 5)))	(TRANSFORM ?C (INPUTS (Population 1) (MetallicElements 2)) (OUTPUTS (Population 1) (MetallicAlloys 1) (MetallicAlloysWaste 1)))	(TRANSFORM ?C (INPUTS (Population 1) (MetallicElements 3) (MetallicAlloys 2)) (OUTPUTS (Population 1) (Electronics 2) (ElectronicsWaste 1)))

A single across-country operator template exists, (TRANSFER ?C_i ?C_k ((?R_1j ?X_1j))), for ?C_i to give ?C_k any amount, ?X_1j, of resource ?R_1j in ?C_i's possession.

World Trade Game Example

Alloys template

((TRANSFORM ?C (INPUTS (R1 1) (R2, 2)) (OUTPUTS (R1 1) (R21, 1) (R21’ 1)),

preconditions are of the form ?ARj <= ?C(?Rj)

Electronics template

(TRANSFORM ?C (INPUTS (R1 3) (R2 2) (R21 2)) (OUTPUTS (R22 2) (R22’ 2) (R1 3)),

preconditions are of the form ?ARj <= ?C(?Rj)

Housing template

(TRANSFORM ?C (INPUTS (R1 5) (R2, 1) (R3 5) (R21 3) (OUTPUTS (R1 5) (R23, 1) (R23’ 1)),

preconditions are of the form ?AIk <= ?C(?Rk)

(TRANSFER ?Cj1 ?Cj2 ((?Ri ?ARi)), where ?ARi <= ?Cj1(?Ri)

A(tlantis)	E(rewon)
R1: 500	R1: 100
R2: 700	R2: 50
R3: 100	R3: 2000
R21: 0	R21: 30
R21’: 0	R21’: 0
R22: 0	R22: 0
R22’: 0	R22’: 0
R23: 0	R23: 0
R23’: 0	R23’: 0

State, n_k

Templates can be used to generate a large number of successors to state n_k.

While an explicit graph representation could have been used with the small robot delivery example, with nontrivial numbers of countries and resources (and possible amounts of resources), an explicit graph is not practical here.

The graph is implicit in the actions and is generated on demand.

World Trade Game Example

A(tlantis)	E(rewon)
R1: 500	R1: 100
R2: 700	R2: 50
R3: 100	R3: 2000
R21: 0	R21: 30
R21’: 0	R21’: 0
R22: 0	R22: 0
R22’: 0	R22’: 0
R23: 0	R23: 0
R23’: 0	R23’: 0

Alloys template

((TRANSFORM ?C (INPUTS (R1 1) (R2, 2)) (OUTPUTS (R1 1) (R21, 1) (R21’ 1)),

preconditions are of the form ?ARj <= ?C(?Rj)

(TRANSFORM A (INPUTS (R1 50*1) (R2, 50*2)) (OUTPUTS (R1 50) (R21, 50) (R21’ 50)),

preconditions 50 <= 500, 100 <= 700

Electronics template

(TRANSFORM ?C (INPUTS (R1 3) (R2 2) (R21 2)) (OUTPUTS (R22 2) (R22’ 2) (R1 3)),

preconditions are of the form ?ARj <= ?C(?Rj)

(TRANSFORM A (INPUTS (R1 30) (R2 20) (R21 20)) (OUTPUTS (R22 20) (R22’ 20) (R1 30)),

preconditions 30 <= 500, 20 <= 700, **20 !<= 0**

Housing template

(TRANSFORM ?C (INPUTS (R1 5) (R2, 1) (R3 5) (R21 3) (OUTPUTS (R1 5) (R23, 1) (R23’ 1)),

preconditions are of the form ?AIk <= ?C(?Rk)

(TRANSFORM E (INPUTS (R1 10*5) (R2, 10*1) (R3 10*5) (R21 10*3) (OUTPUTS (R1 10*5) (R23, 10*1) (R23’ 10*1)),

preconditions are of the form 50 <= 100, 10 <= 50, 50 <= 2000, 30 <= 30

(TRANSFER ?Cj1 ?Cj2 ((?Ri ?ARi)), where ?ARi <= ?Cj1(?Ri)

(TRANSFER E A ((R3 500)), preconditions 500 <= 2000

World Trade Game Example

A(tlantis)
R1: 500
R2: 700
R3: 100
R21: 0
R21’: 0
R22: 0
R22’: 0
R23: 0
R23’: 0

E(rewon)
R1: 100
R2: 50
R3: 2000
R21: 30
R21’: 0
R22: 0
R22’: 0
R23: 0
R23’: 0

Alloys template (one instantiation)

(TRANSFORM A (INPUTS (R1 50*1) (R2, 50*2)) (OUTPUTS (R1 50) (R21, 50) (R21’ 50)),

preconditions 50 <= 500, 100 <= 700

A(tlantis)	E(rewon)
R1: 500	R1: 100
R2: 600	R2: 50
R3: 100	R3: 2000
R21: 50	R21: 30
R21’: 50	R21’: 0
…	…

Electronics template (one instantiation)

(TRANSFORM A (INPUTS (R1 30) (R2 20) (R21 20)) (OUTPUTS (R22 20) (R22’ 20) (R1 30)),

preconditions 30 <= 500, 20 <= 700, **20 !<= 0. (insufficient resources)**

No successor
X

Housing template (one instantiation)

(TRANSFORM E (INPUTS (R1 10*5) (R2, 10*1) (R3 10*5) (R21 10*3) (OUTPUTS (R1 10*5) (R23, 10*1) (R23’ 10*1)),

preconditions are of the form 50 <= 100, 10 <= 50, 50 <= 2000, 30 <= 30

A(tlantis)	E(rewon)
R1: 500	R1: 10
R2: 700	R2: 40
R3: 100	R3: 1950
R21: 0	R21: 0
R21’: 0	R21’: 0
R22: 0	R22: 0
R22’: 0	R22’: 0
R23: 0	R23: 10
R23’: 0	R23’: 10

Transfer (one instantiation)
(TRANSFER E A ((R3 500)), preconditions 500 <= 2000

A(tlantis)	E(rewon)
R1: 500	R1: 100
R2: 700	R2: 50
R3: 600	R3: 1500
R21: 0	R21: 30
R21’: 0	R21’: 0
…	…

Douglas H. Fisher

The Generic Algorithm for Searching Implicit Graphs

Exploring Alternatives With Search

Douglas H. Fisher

A Revision to Generic (Heuristic) Search Algorithm for Implicit Graphs

structure *SearchNode (State Parent Action Path-Cost DistEst, Children)*

SearchNode Search (~~Vertices V~~, ~~Arcs~~ Actions A, S_0, Goal Condition G HeuristicFn H)

/* … **assume that each entry in A, a, now includes an operator a.op and cost a.cost;**

G is a Boolean goal condition */

SearchNode N = new SearchNode(State S_0, Parent NULL, Action NULL, Path-Cost 0, **DistEst H(S_0, G)**, Children NULL)

Frontier = [N]

Reached = {N}

while Frontier != [ ] do

select and remove N from Frontier

if N.State satisfies G then return N // from which the path from S_0 to N.State can be recovered
Otherwise, generate successors of N (next slide).

return 〈〉

A Revision to Generic (Heuristic) Search Algorithm for Implicit Graphs (cont.)

structure ***SearchNode (State Parent Action Path-Cost DistEst, Children)***

SearchNode Search (~~Vertices V~~, ~~Arcs~~ Actions A, S_0, Goal Condition G HeuristicFn H)

… (previous slide)

    if N.State satisfies G then return N   // from which the path from S_0 to N.State can be recovered
    for each action a in A that is applicable to N.State
        SearchNode L = new SearchNode(State **Apply(a.op, N.State)**, Parent N, Action a,
                                      Path-Cost N.Path-Cost + a.cost, **DistEst default**,
                                      Children NULL)
        L.DistEst = H(L.State, G)
        if !exists Node M in Reached s.t. M.State == L.State or L.Path-Cost < M.Path-Cost
            N.Children = N.Children + L
            Reached = Reached - M + L
            Frontier = Frontier + L
  return 〈〉

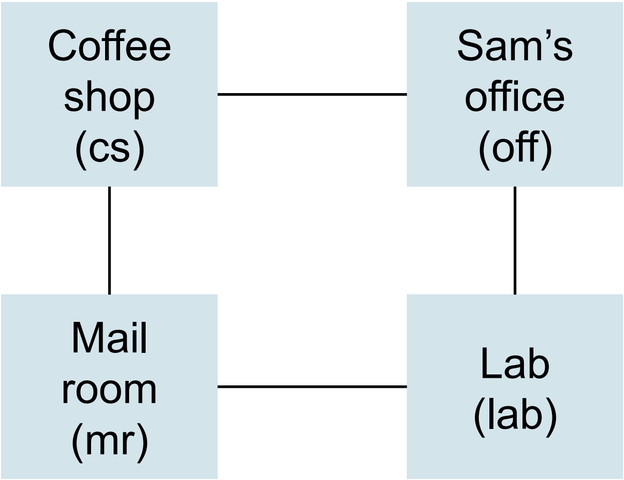
Generate Successors (N, A)

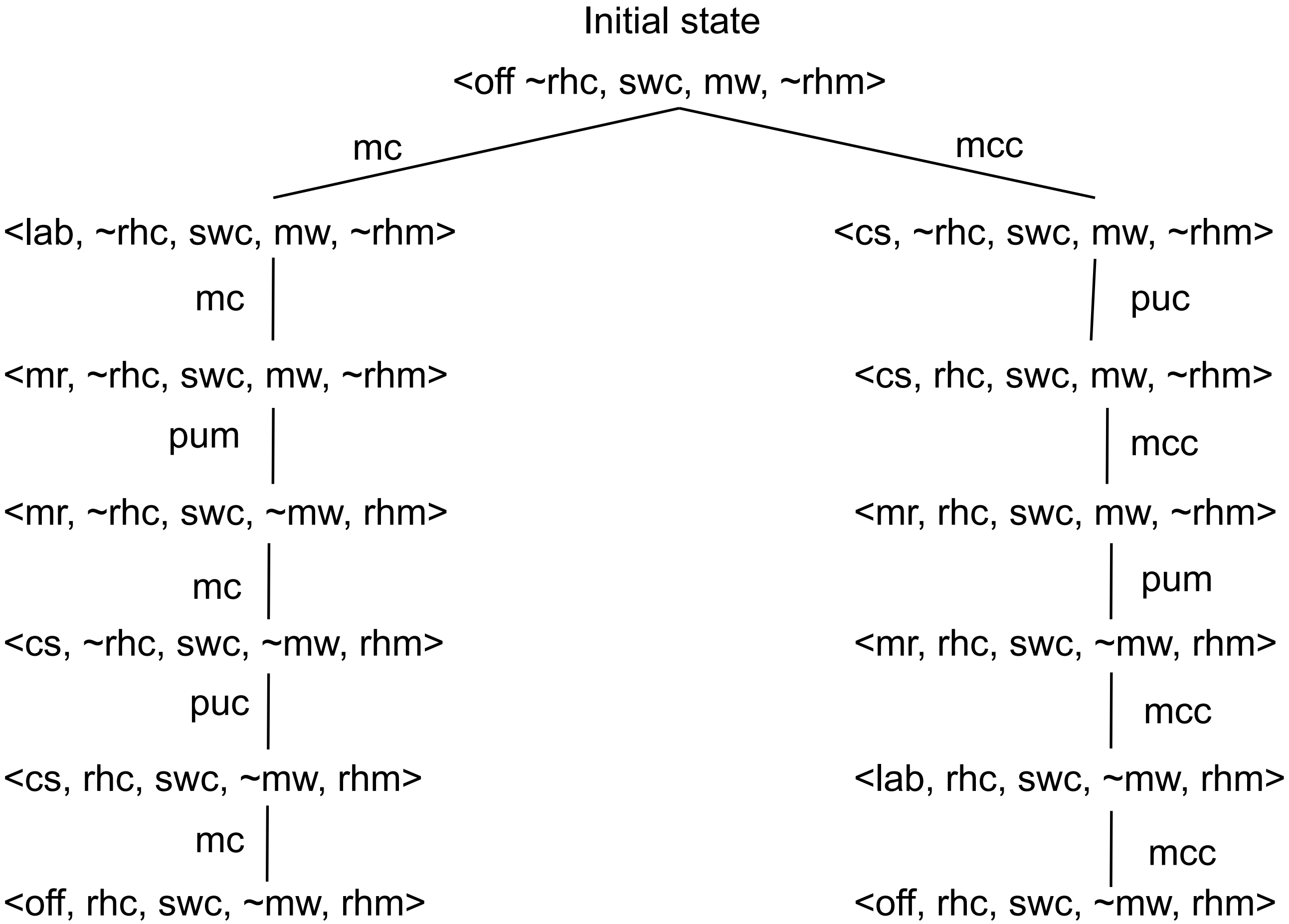
This is no longer a simple check of a reencountered atomic and extant vertex but now requires a check to see if the two are exact copies (of factored or structured representations).

Douglas H. Fisher

Delivery Robot Example of Redundant Paths With Implicit Graphs

- rloc (Rob's location) is four-valued.
- rhc (Rob has coffee) is binary-valued.
- swc (Sam wants coffee) is binary-valued.
- mw (mail waiting) is binary-valued.
- rhm (Rob has mail) is binary-valued.





Redundant paths assuming state copy equality

The End