CMSC 170

Introduction to Artificial Intelligence

CNM Peralta

2nd Semester AY 2014-2015

PROBLEM SOLVING

PRELIMINARIES

For the time being, we will be considering **fully-observable environments** only.

PRELIMINARIES

Problem complexity will depend on the **stochasticity** and **discreteness** of the environment.

PRELIMINARIES

The **set of all possible states** is the *state space, S*.

Problem solving

The theory and technology of building agents that can **plan ahead** to **solve problems**.

WHAT IS A PROBLEM?

Problems are broken down into several components.

1.

Initial state, s_0

The **state** of the **agent** and its **environment** at the **beginning** of the problem.

$$s_0 \in S$$

2.

`Actions(s)`

A function that takes a **state**, s, as input and returns the **list of possible actions**.

$$Actions(s) \rightarrow \{a_{1,}\, a_{2,}\, a_{3,} \ldots\}$$

3.

`Result(s, a)`

A function that takes a **state**, s, and **action**, a, as input and **returns** the **next state**, s'.

$$Result(s, a) \rightarrow s'$$

4.

`GoalTest(s, a)`

A function that takes a **state**, s, as input and **returns** `true` if it is the **goal**, `false` **otherwise**.

$$GoalTest \rightarrow T | F$$

5.

`PathCost(s`$\overset{a}{\to}$`s`$\overset{a}{\to}$`s`$\overset{a}{\to}$`...)`

A function that takes a **sequence of state-action transitions**, called the **path**, and **returns** the **cost of that path**.

$$PathCost\left(s\overset{a}{\to}s\overset{a}{\to}s\overset{a}{\to}\ldots\right)\to n$$

5. (CONT.)

Path costs are usually **additive**, that is, it is the **sum of the cost of individual steps**, requiring a

step-cost function.

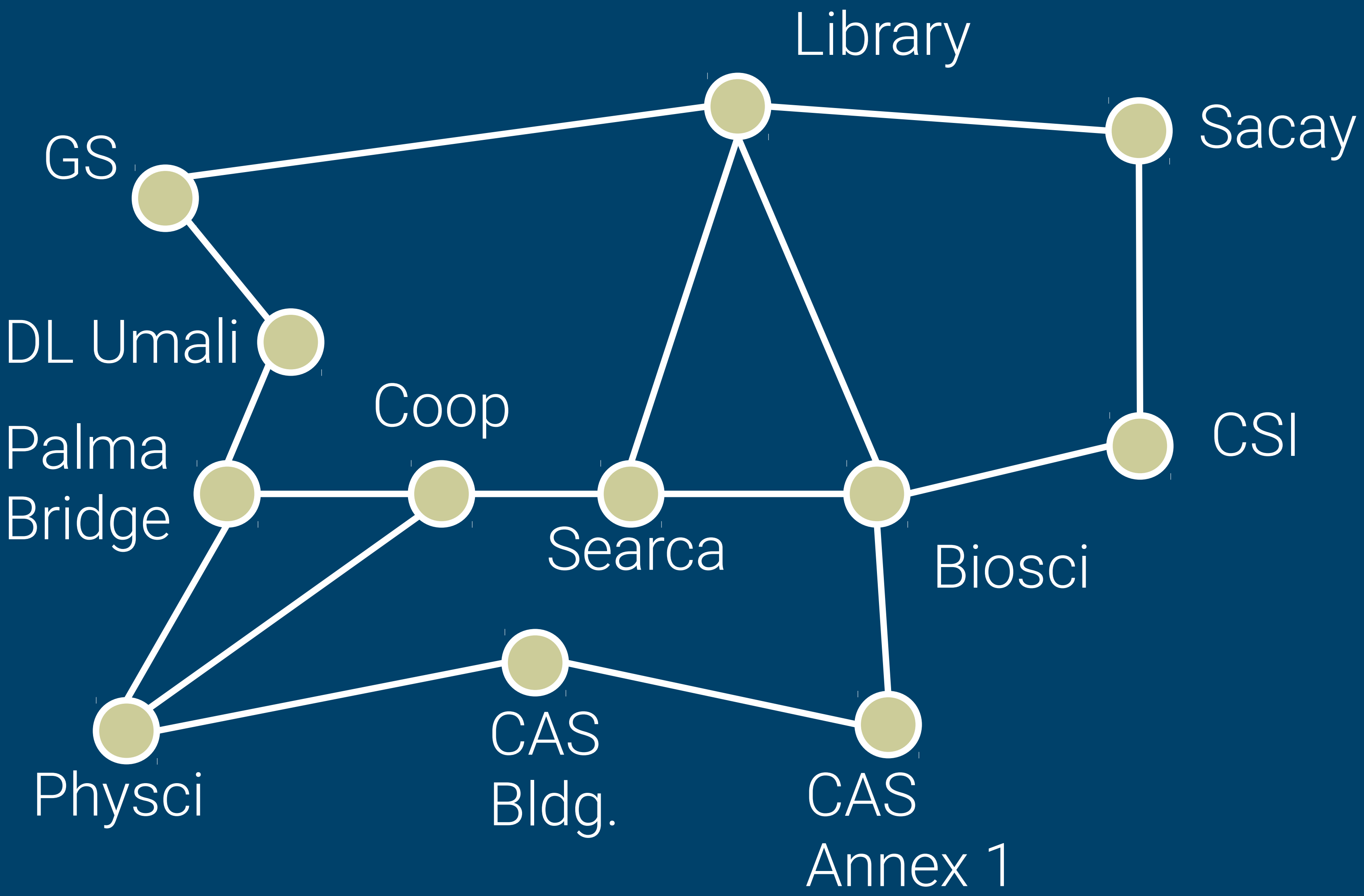
5. (CONT.)

`StepCost(s,a,s')`

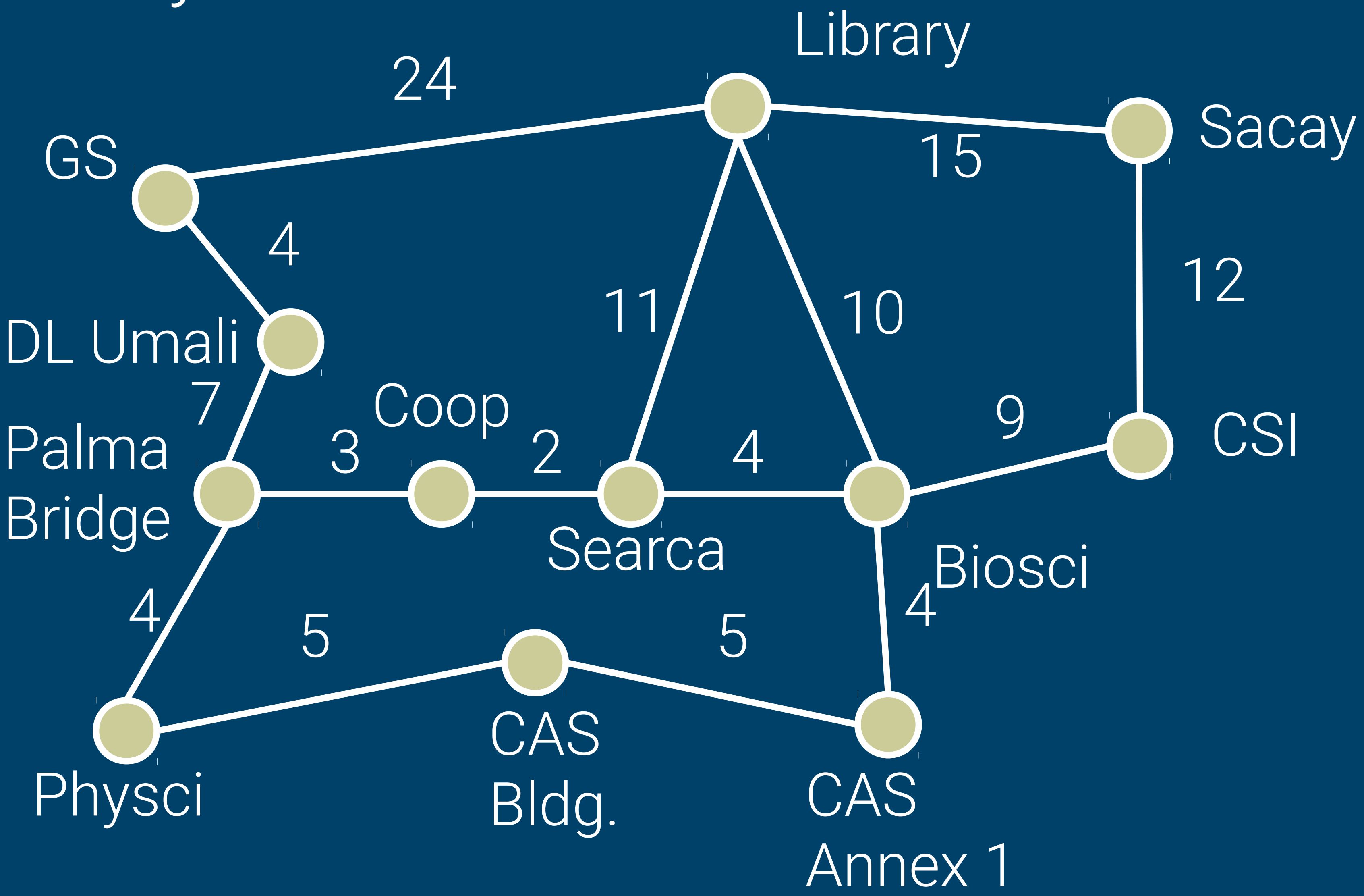
A function that takes a **state**, s, **action**, a, and **next state**, s', and **returns** the **cost of that step**.

$$StepCost(s, a, s') \rightarrow n$$

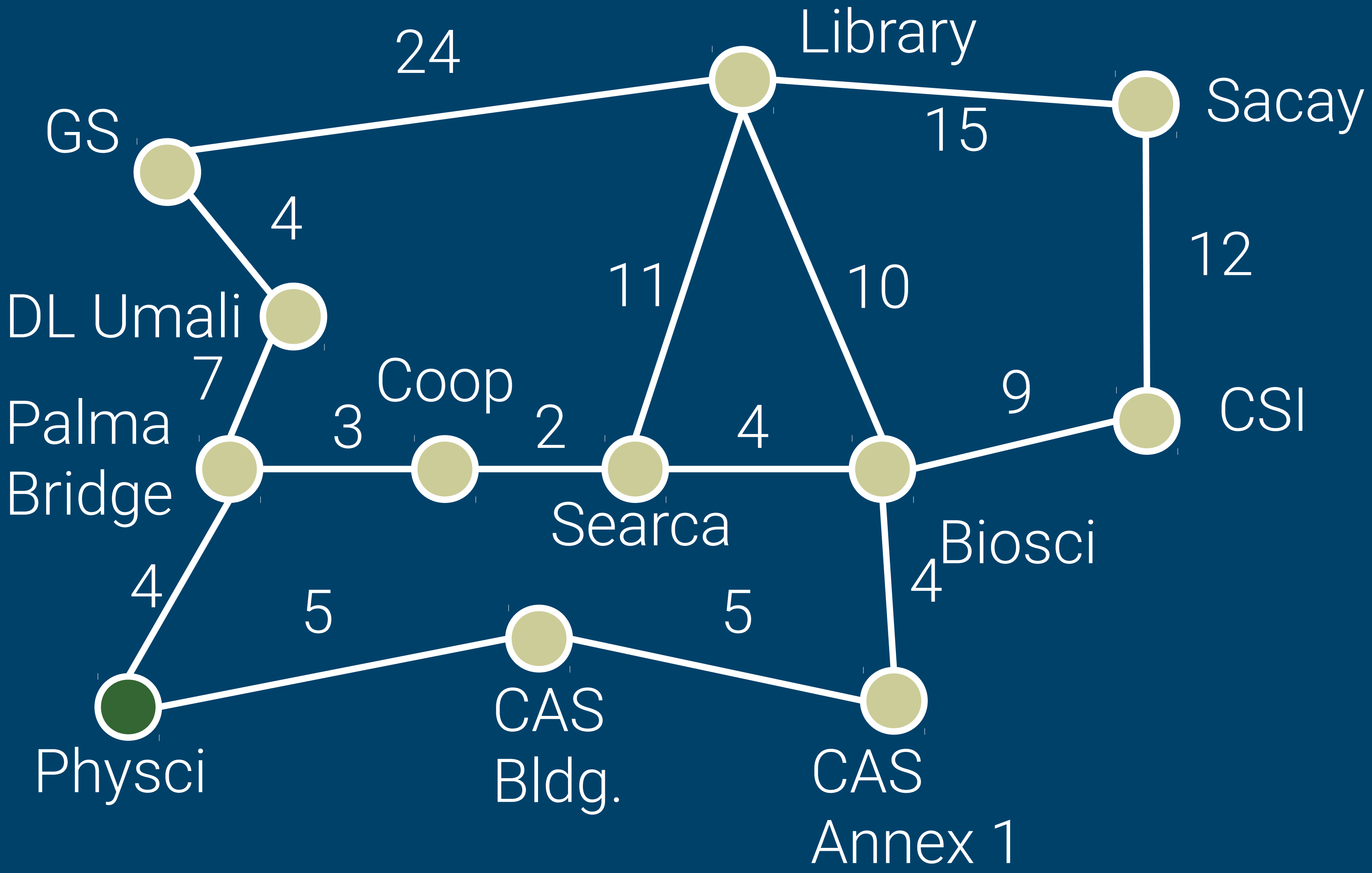
EXAMPLE: ROUTE-FINDING



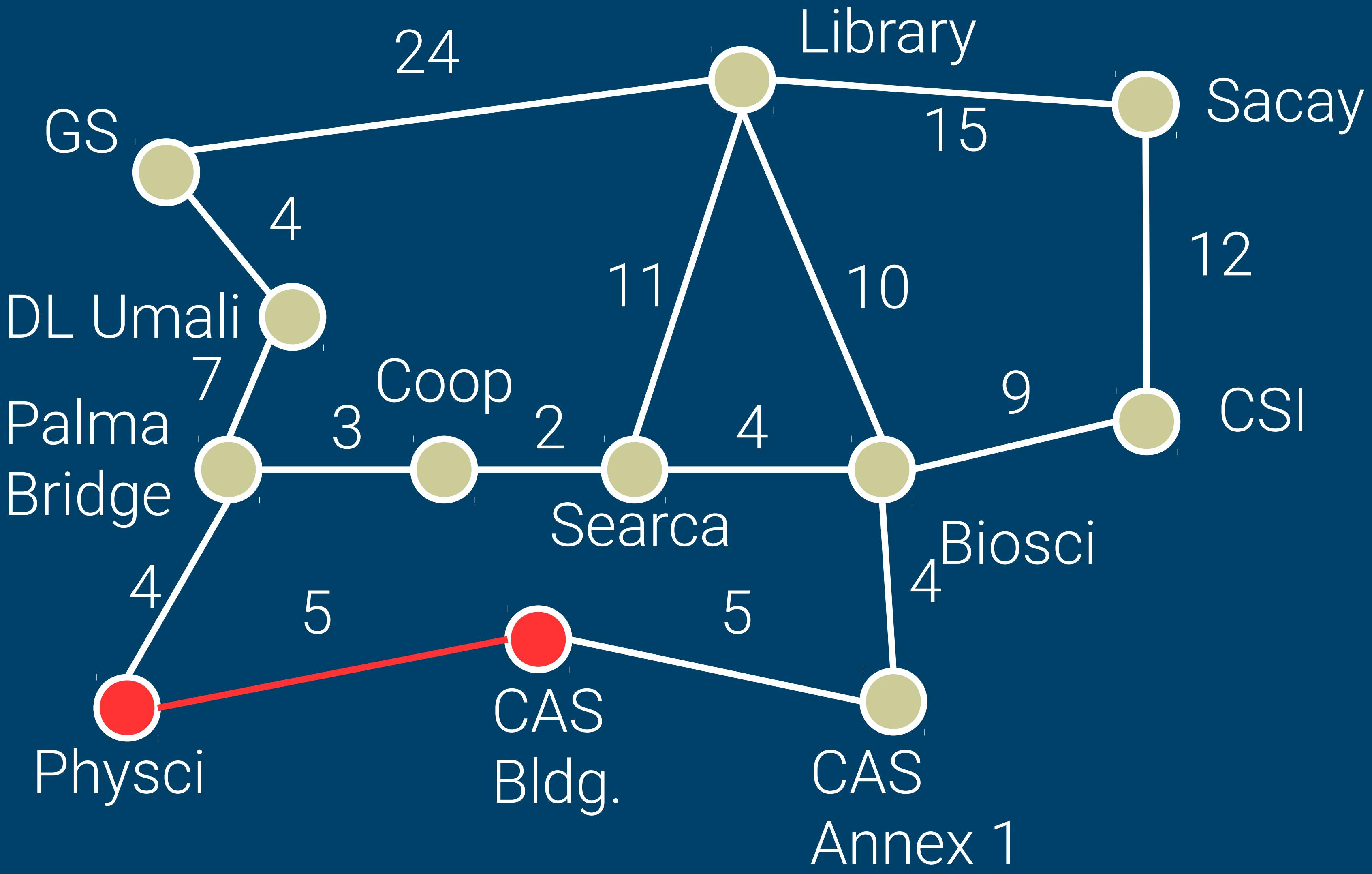
What route should one take from Physci to the Library?



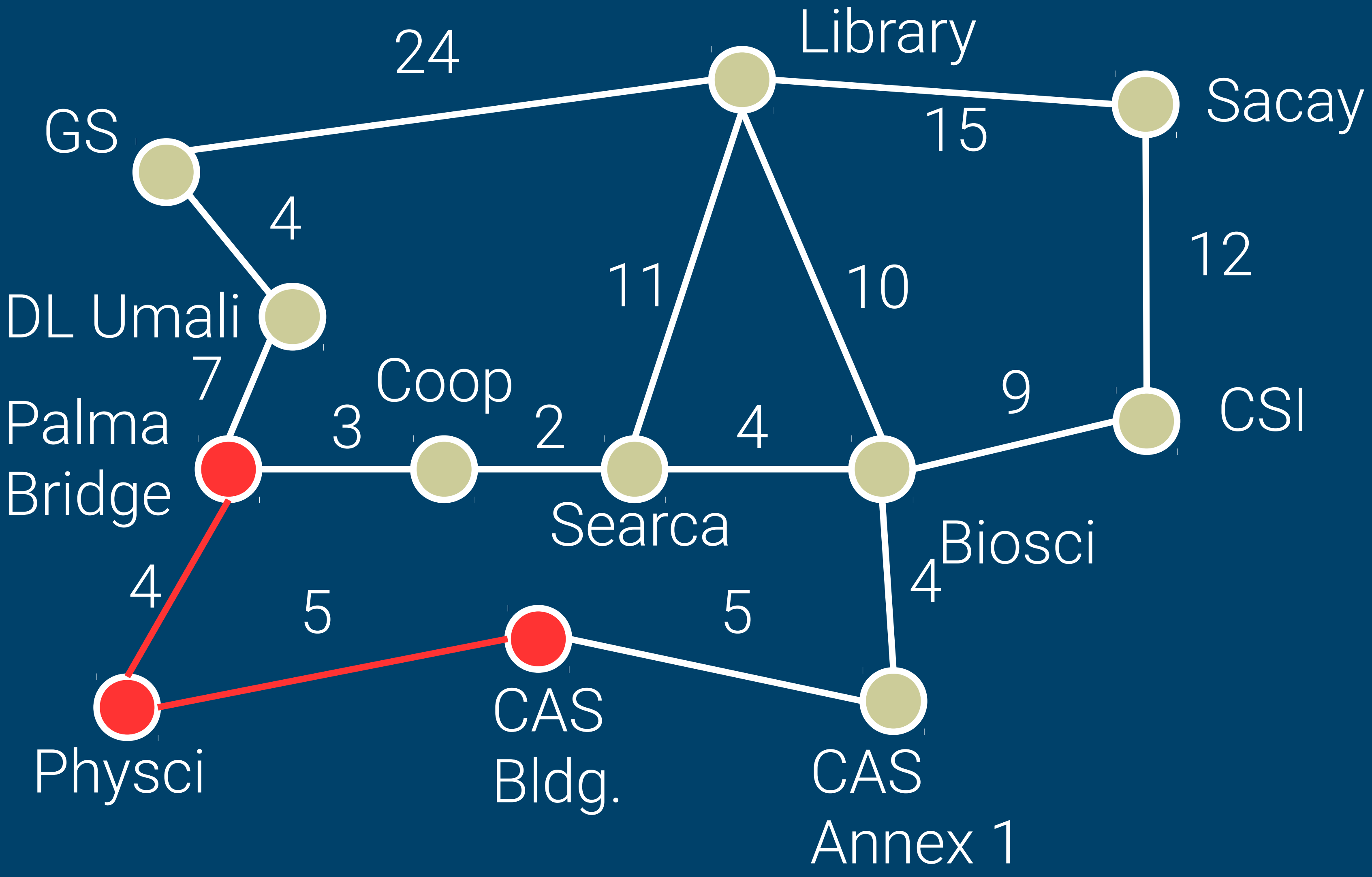
The initial state has the **agent at Physci**, and **all other nodes are unexplored**.



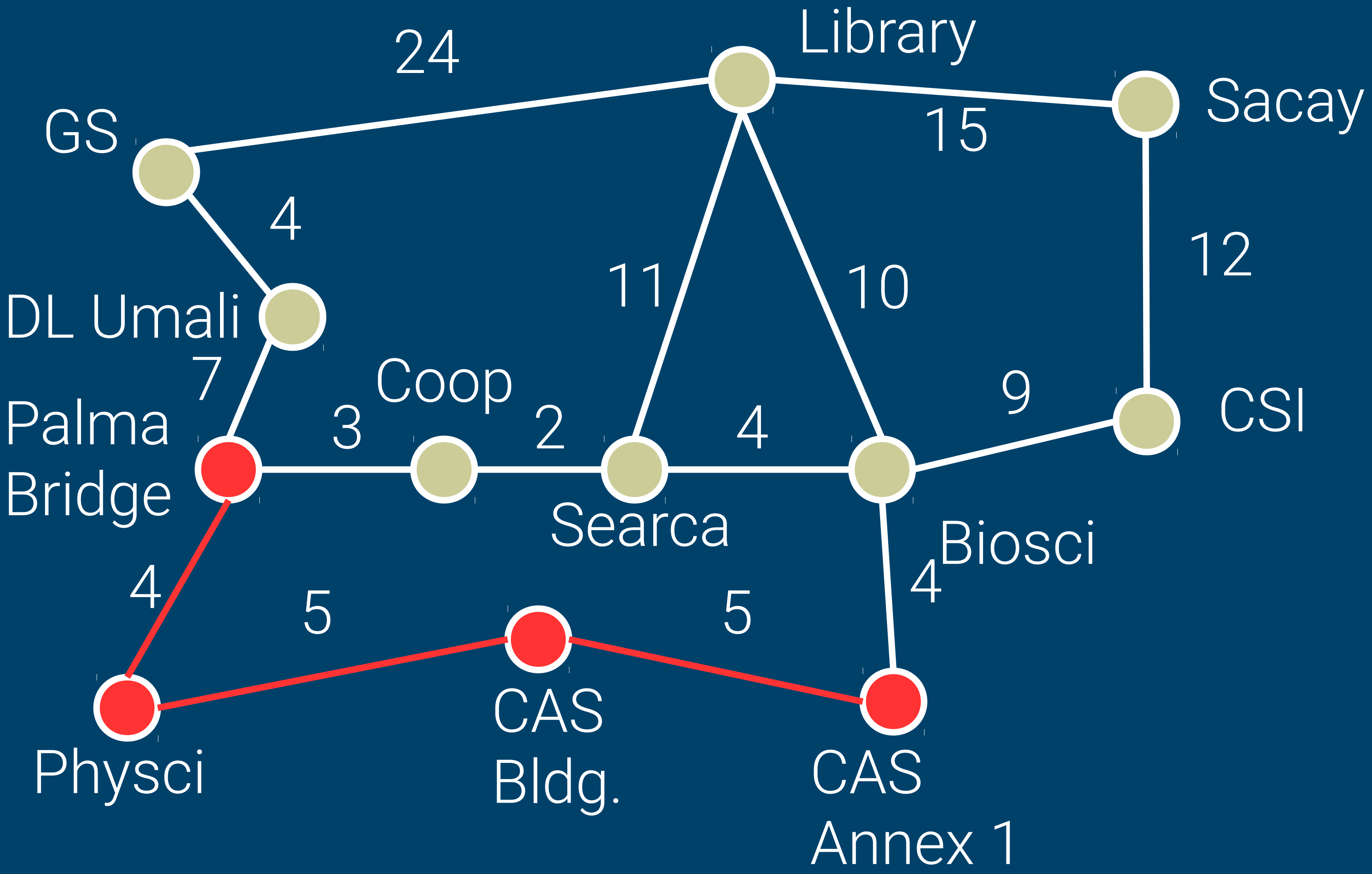
The agent **explores** the graph and **remembers** the paths it has explored.



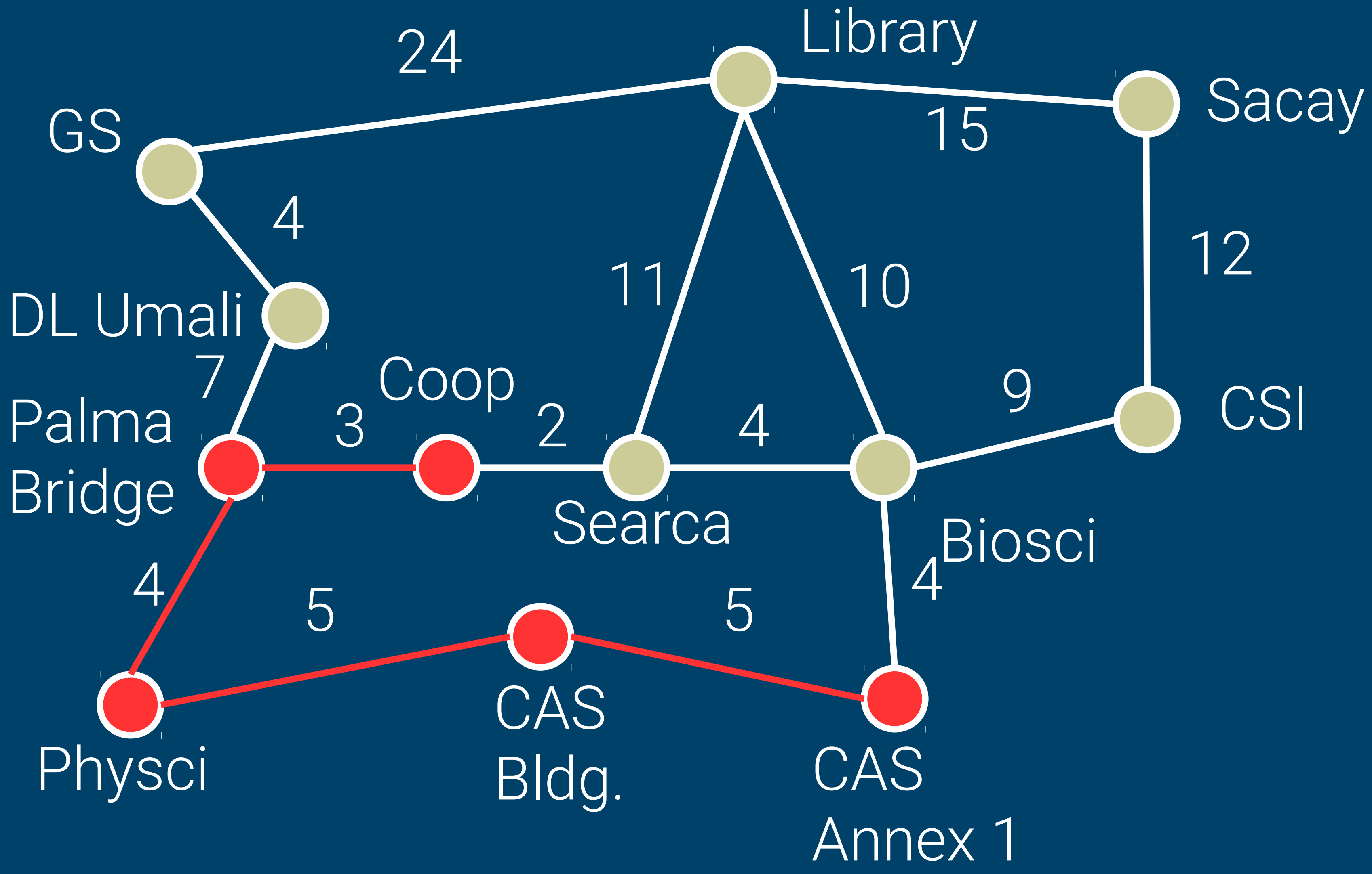
The agent **explores** the graph and **remembers** the paths it has explored.



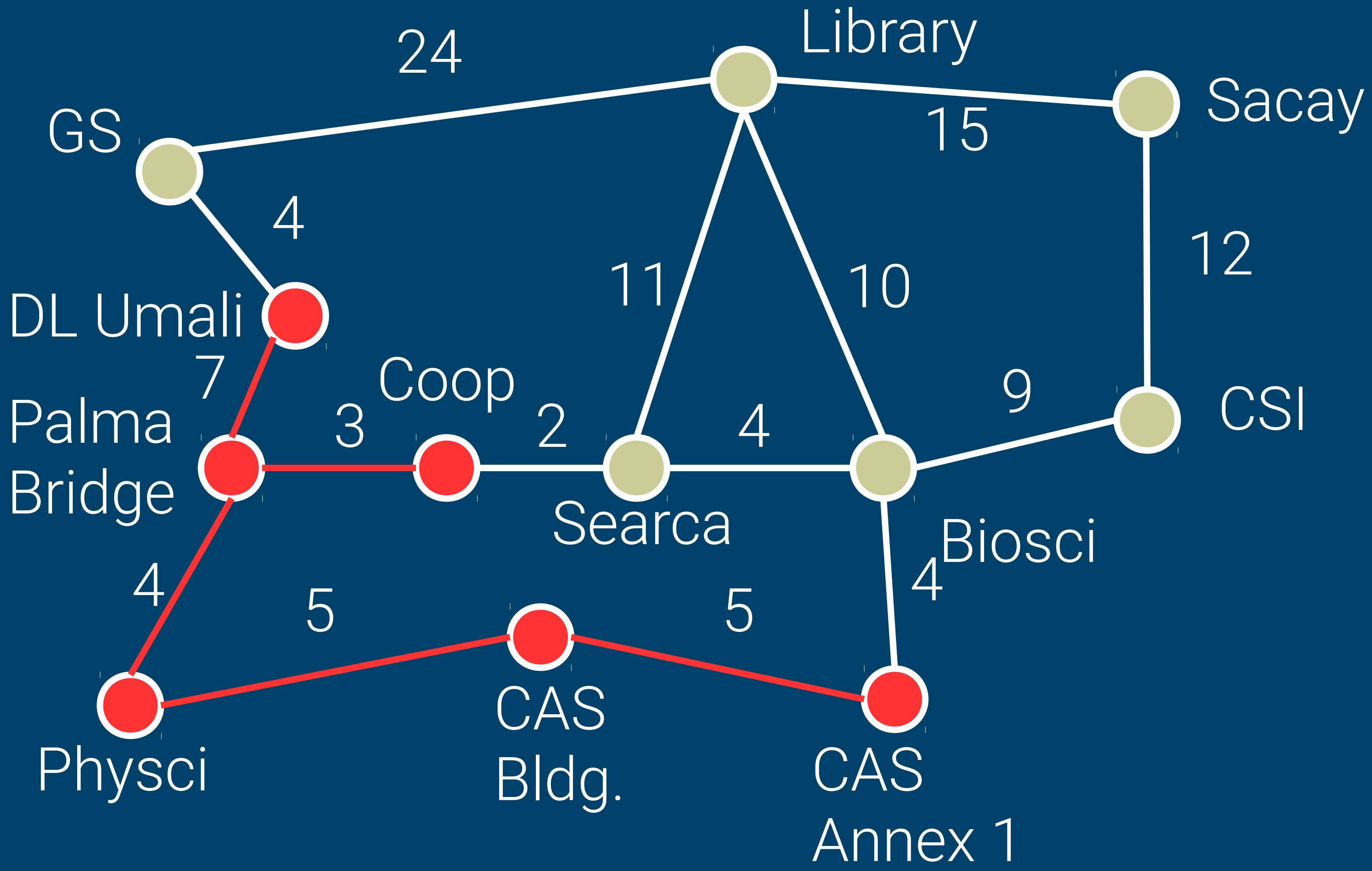
The agent **explores** the graph and **remembers** the paths it has explored.



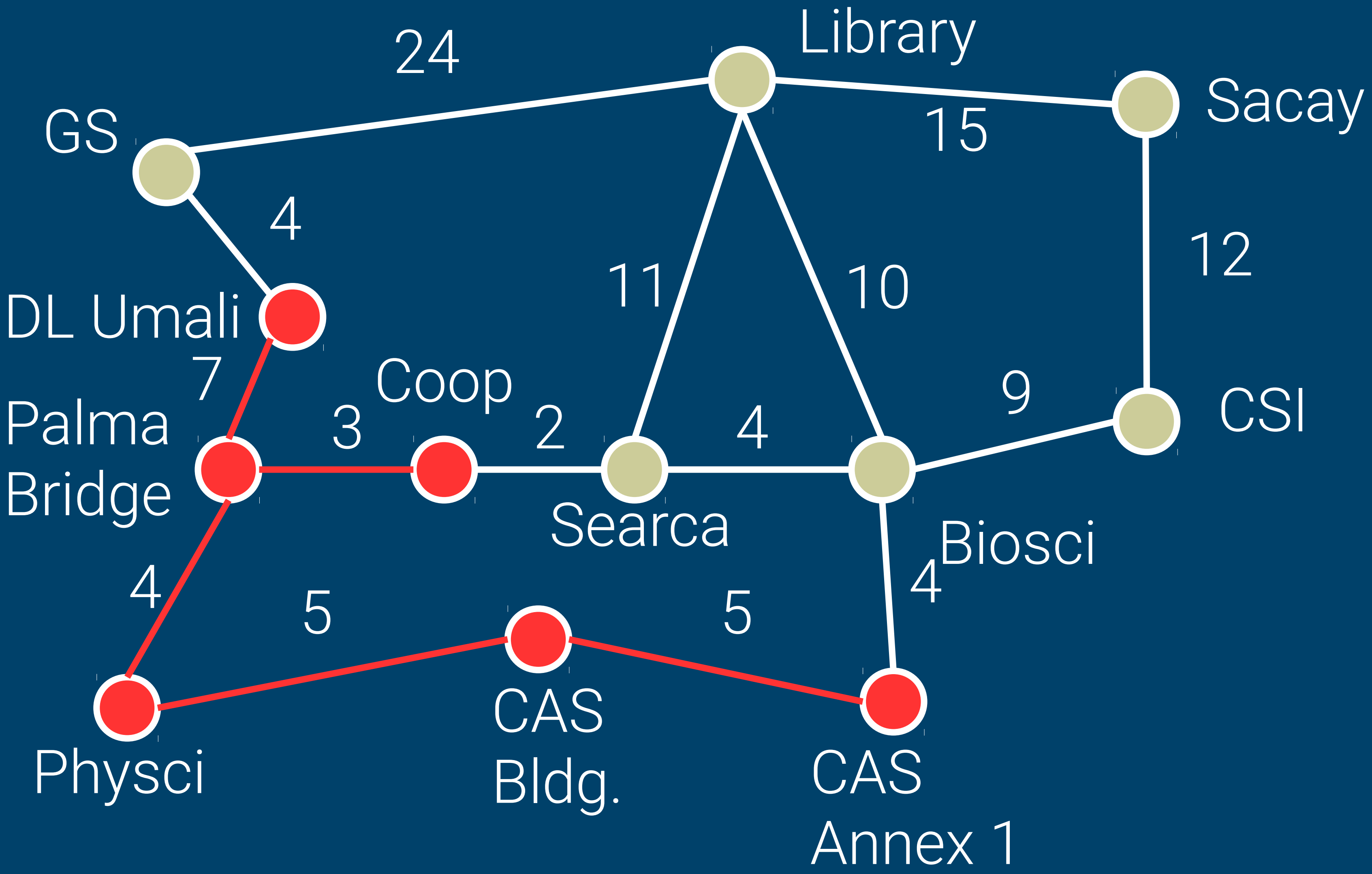
The agent **explores** the graph and **remembers** the paths it has explored.



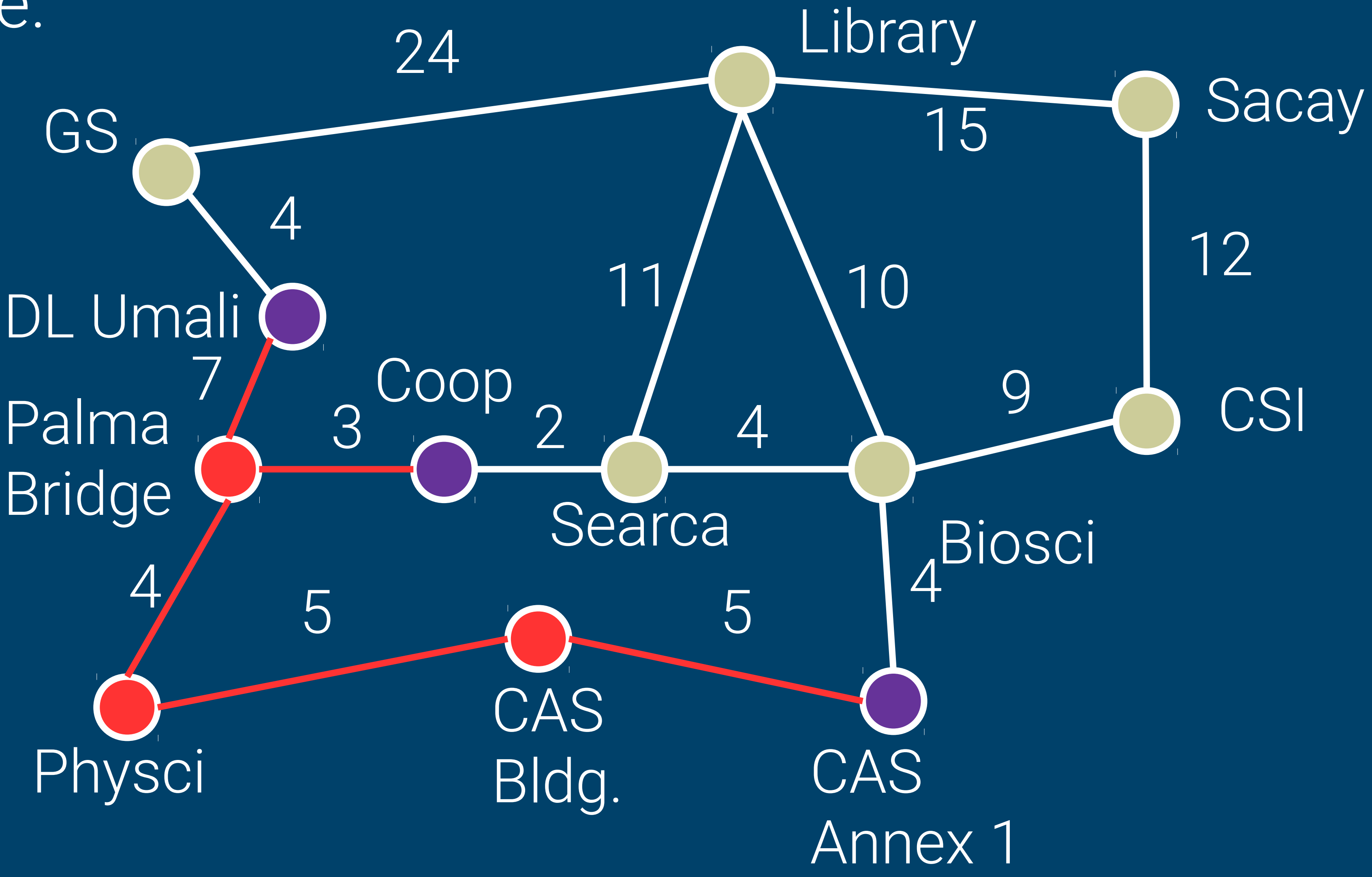
The agent **explores** the graph and **remembers** the paths it has explored.



At each point during exploration, the **state space** can be divided into **three parts**.



The **frontier** (purple) divides the **explored** (light brown) and **unexplored** (red) regions of the state space.



TREE SEARCH ALGORITHMS

Tree Search

Family of algorithms that **superimpose** a **search tree** on the **state space** of the problem.

```
function treeSearch(problem) {
  frontier={[initial]}
  while(frontier is not empty) {
    path=removeChoice(frontier)
    s=path.end
    if(GoalTest(s)) return path
    for (a in Actions(s)) //expand path
      frontier.add(path + $s \xrightarrow{a}$ Result(s, a))
  }
}
```

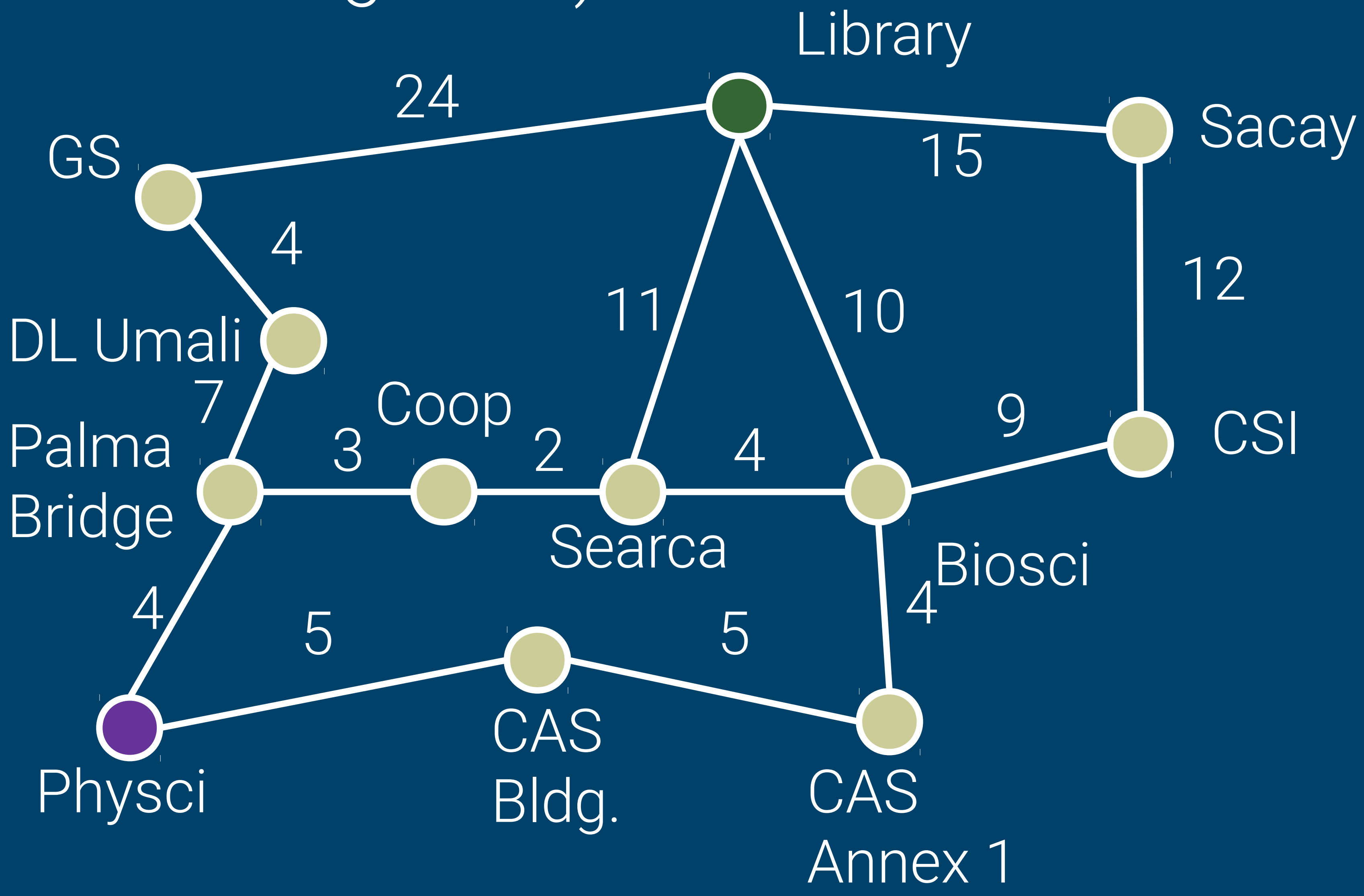
Tree search algorithms vary in their **choice of the path to explore next** in the frontier.

```
function treeSearch(problem) {
  frontier={[initial]}
  while(frontier is not empty) {
    path=removeChoice(frontier)
    s=path.end
    if(GoalTest(s)) return path
    for (a in Actions(s)) //expand path
      frontier.add(path + s $\xrightarrow{a}$ Result(s, a))
  }
}
```

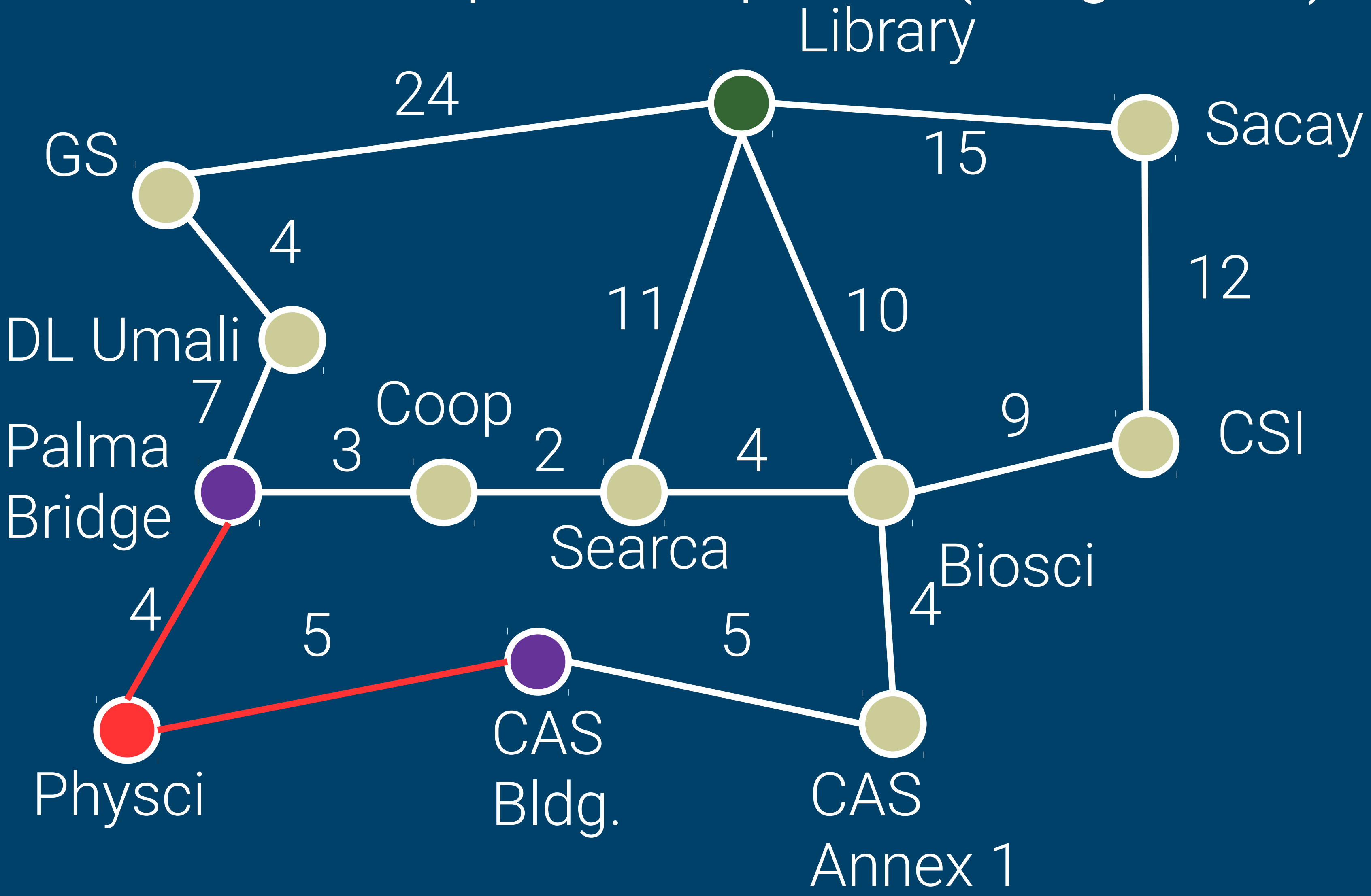
Breadth-First Search

Tree search algorithm that chooses the **path with the shortest length** (not cost) among unexplored paths in the frontier.

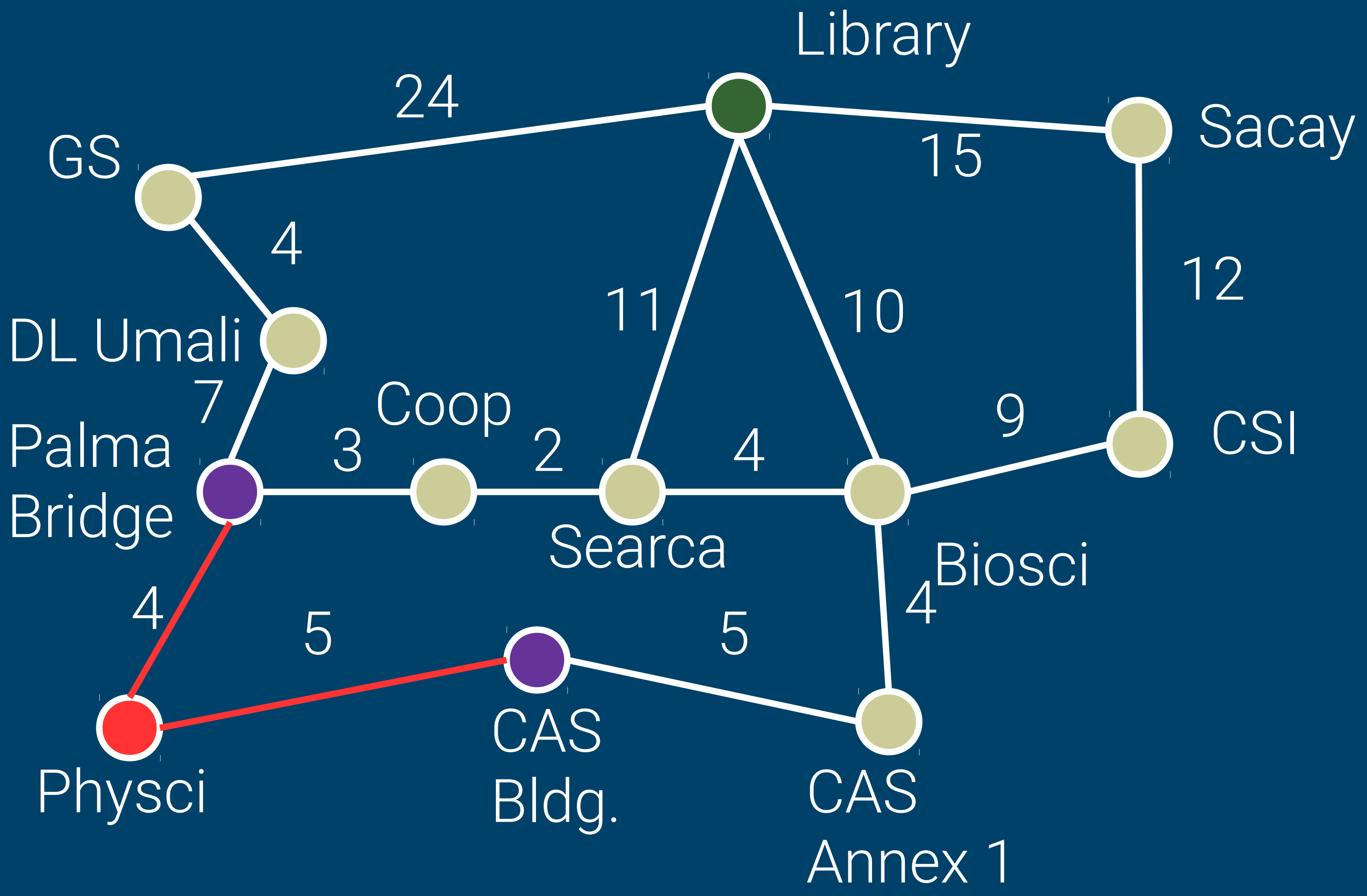
EXAMPLE. Add the initial state first (path length = 0).



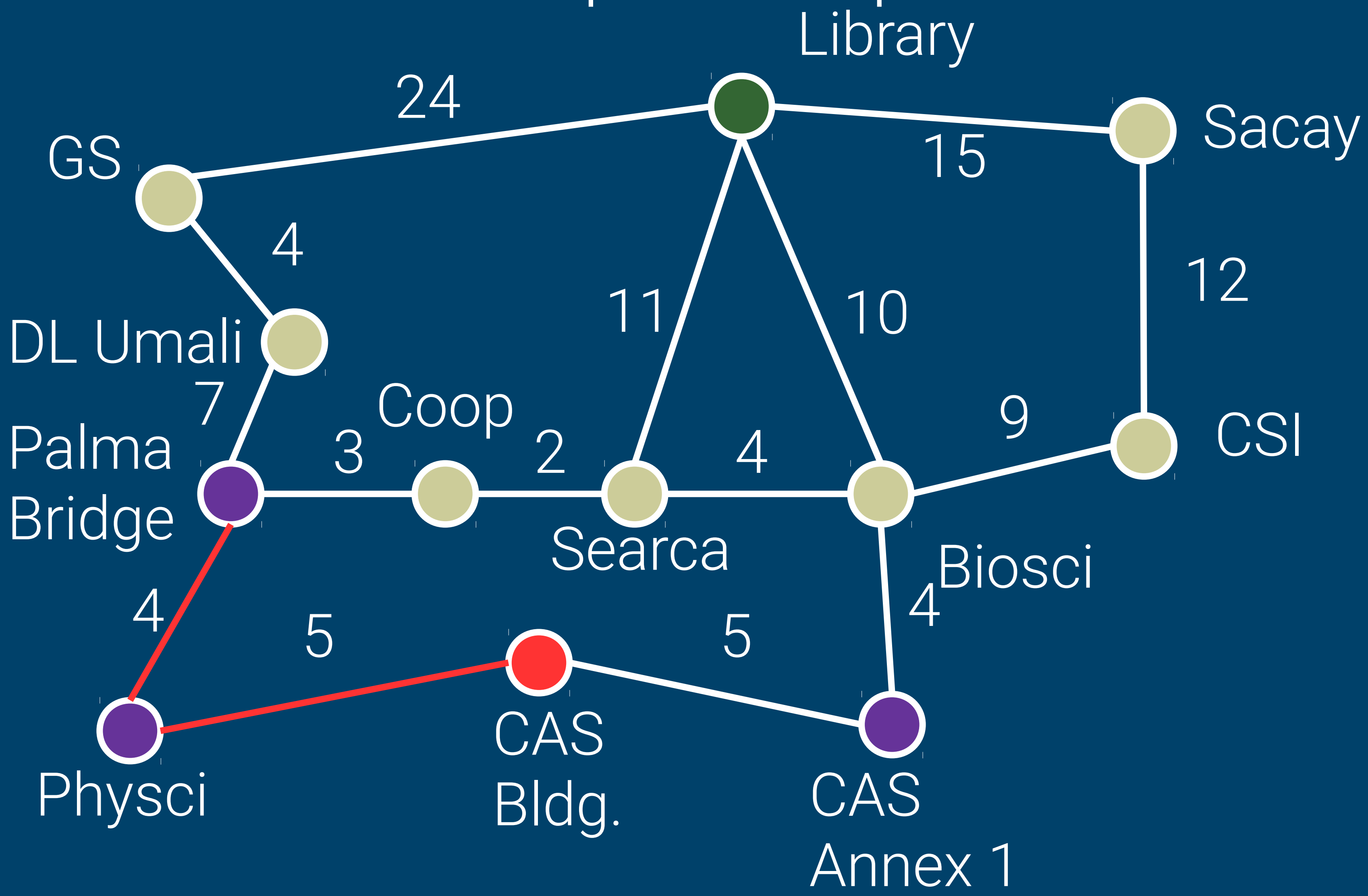
EXAMPLE. Remove Physci from frontier and add expanded paths (length = 1).



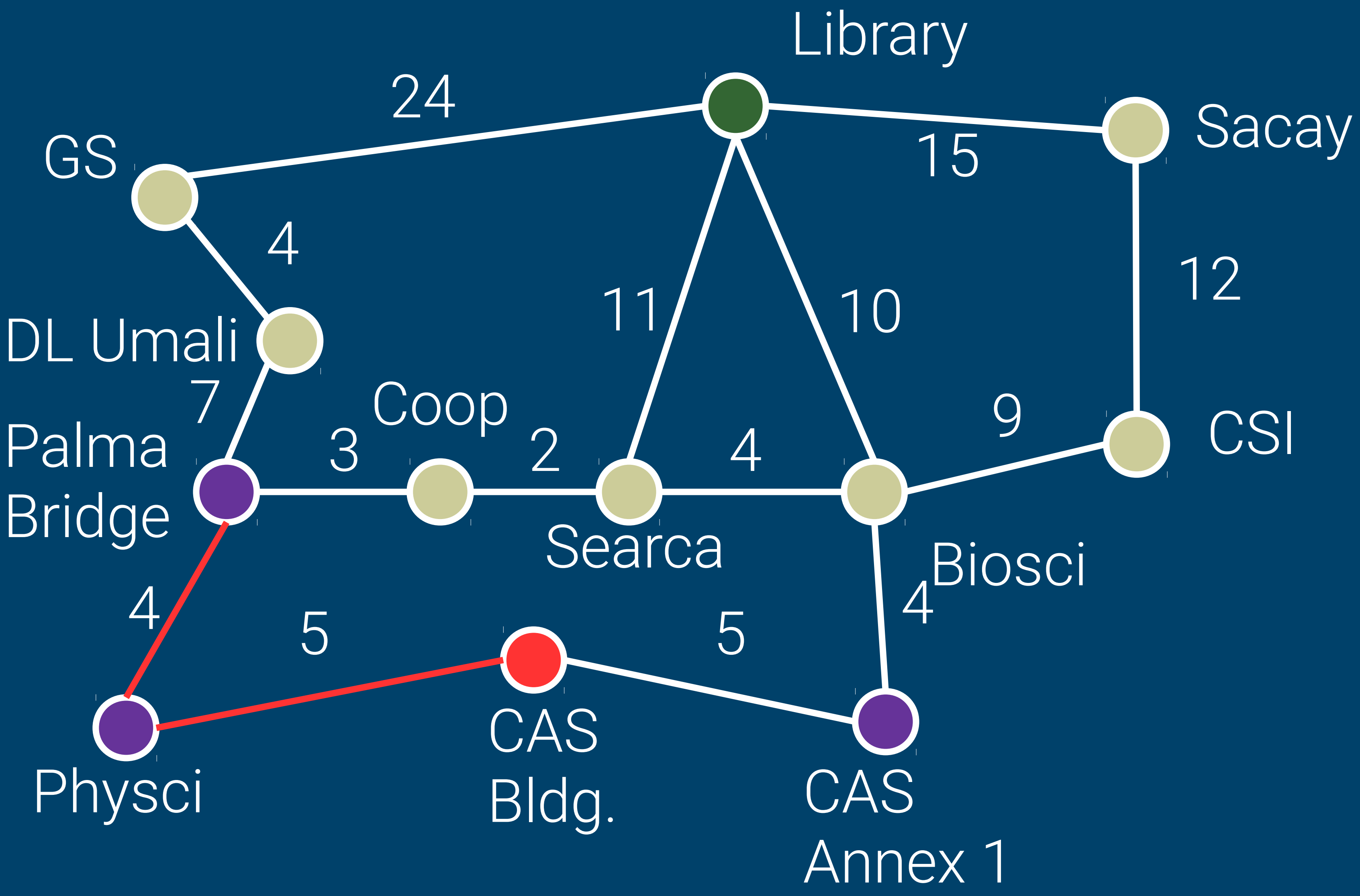
EXAMPLE. Break ties randomly.



EXAMPLE. Say we remove from the frontier, and we expand its path.

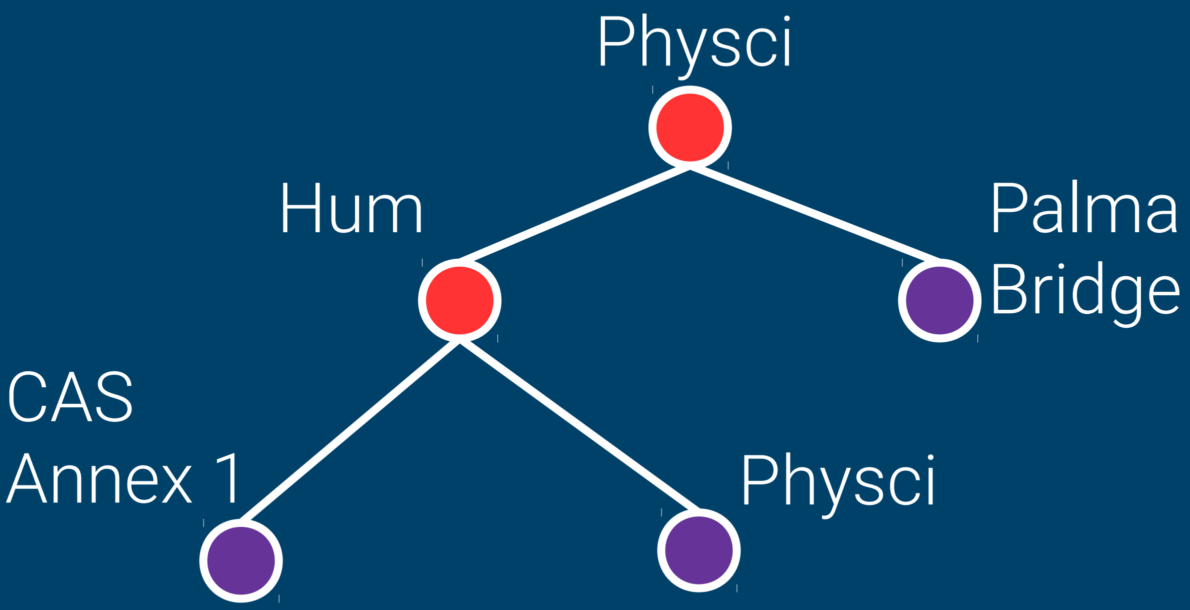


EXAMPLE. We have three paths; one path of length 1 and two paths of length 2.



Tree search, by default, **unwittingly backtracks** to previously visited states.

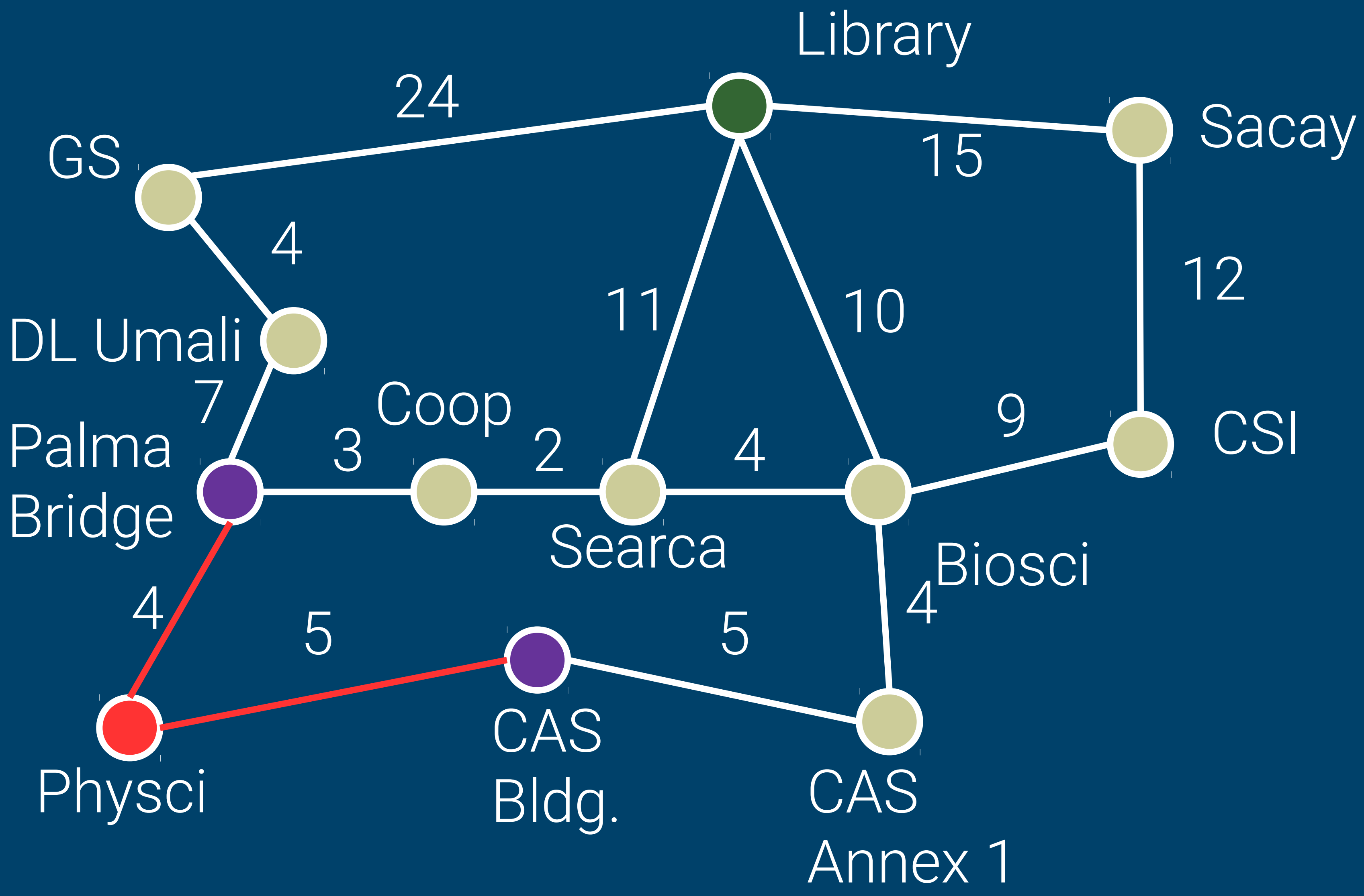
The resulting superimposed tree looks like:



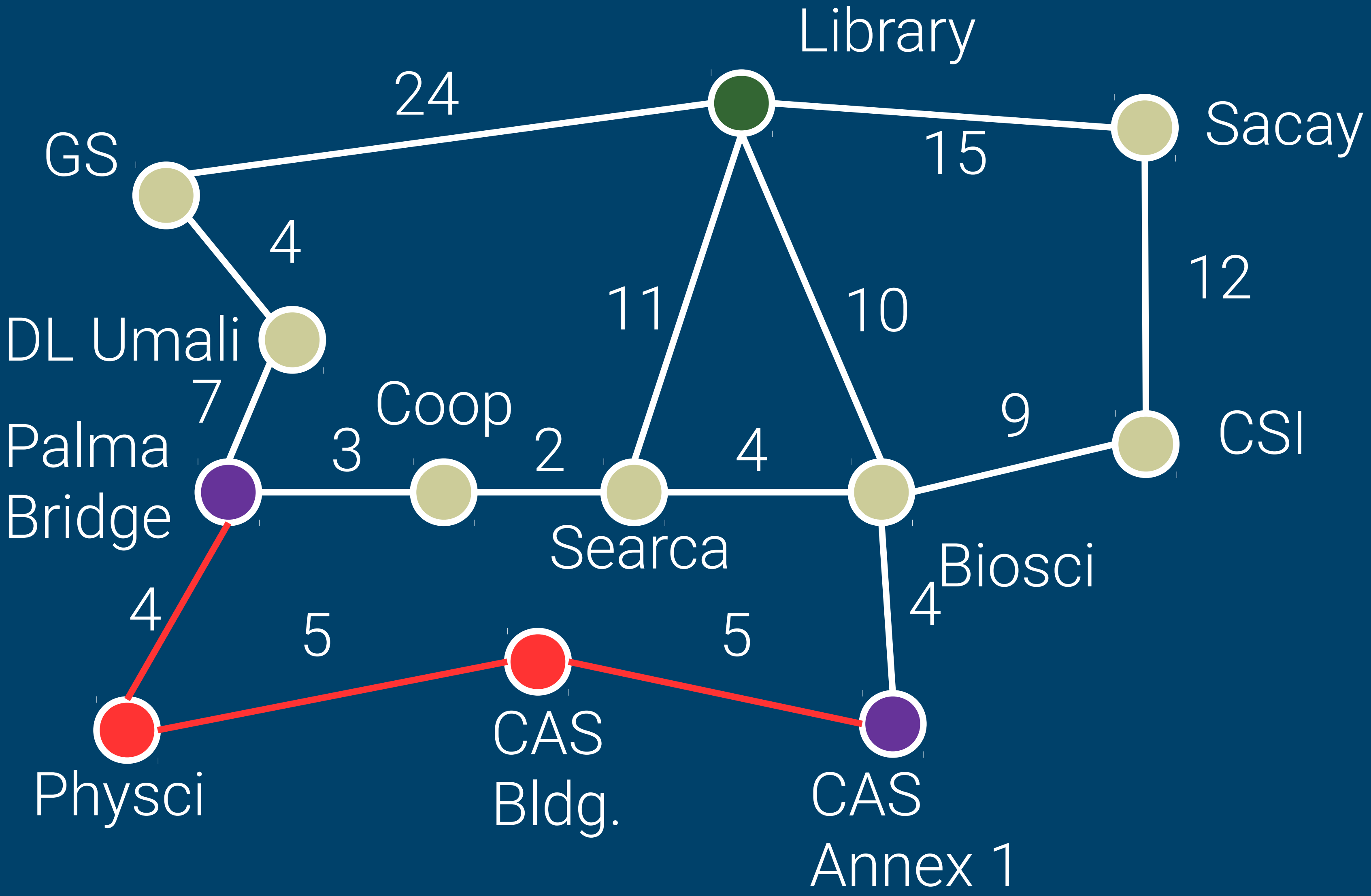
Thus, we need to **keep track** of **previously-visited** states.

```
function graphSearch(problem) {
  frontier={[initial]}  explored={}
  while(frontier is not empty) {
    path=removeChoice(frontier)
    s=path.end  explored.add(s)
    if(GoalTest(s)) return path
    for (a in Actions(s)) //expand path
      if(Result(s,a) ∉ frontier ⋃ explored)
      frontier.add(path + s → Result(s, a))
  }
}
```

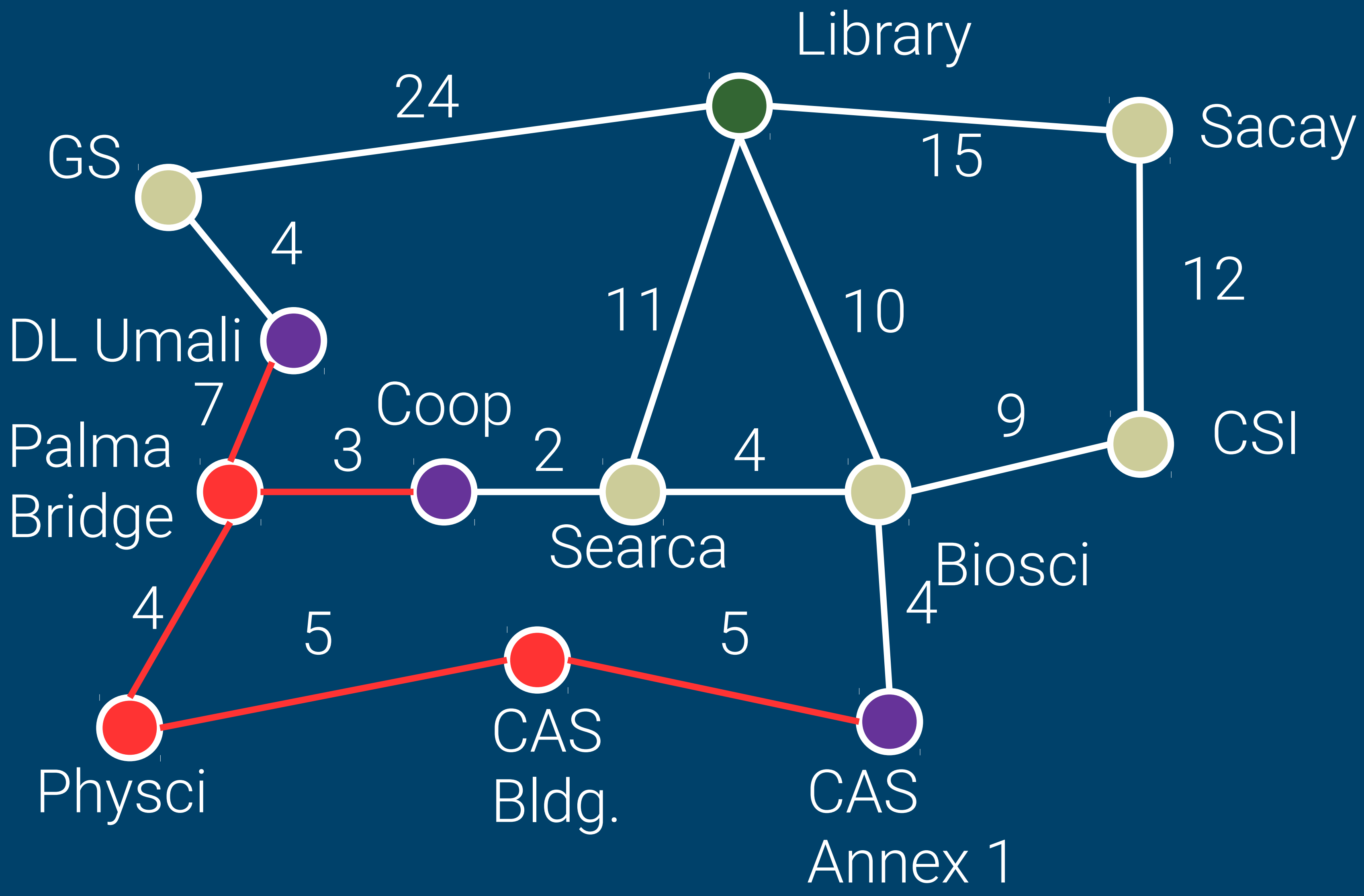
EXAMPLE. This time, when we expand CAS Bldg., we no longer consider Physci.



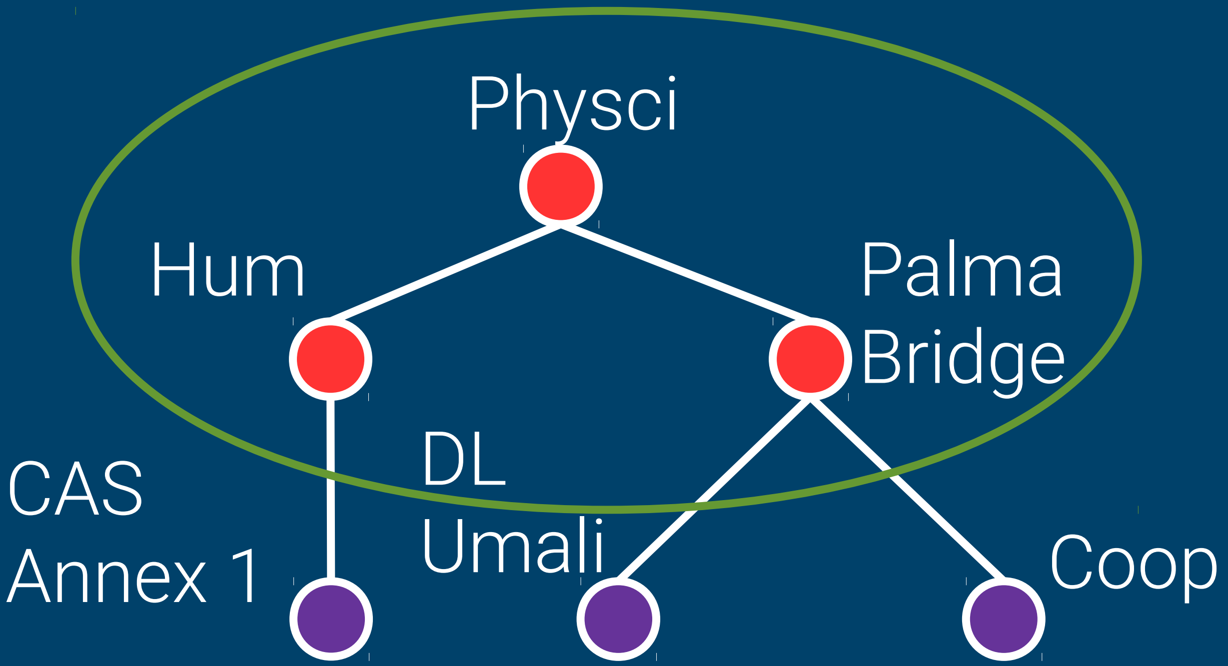
EXAMPLE. This time, when we expand CAS Bldg., we no longer consider Physci.



EXAMPLE. Expand the path of length 1.

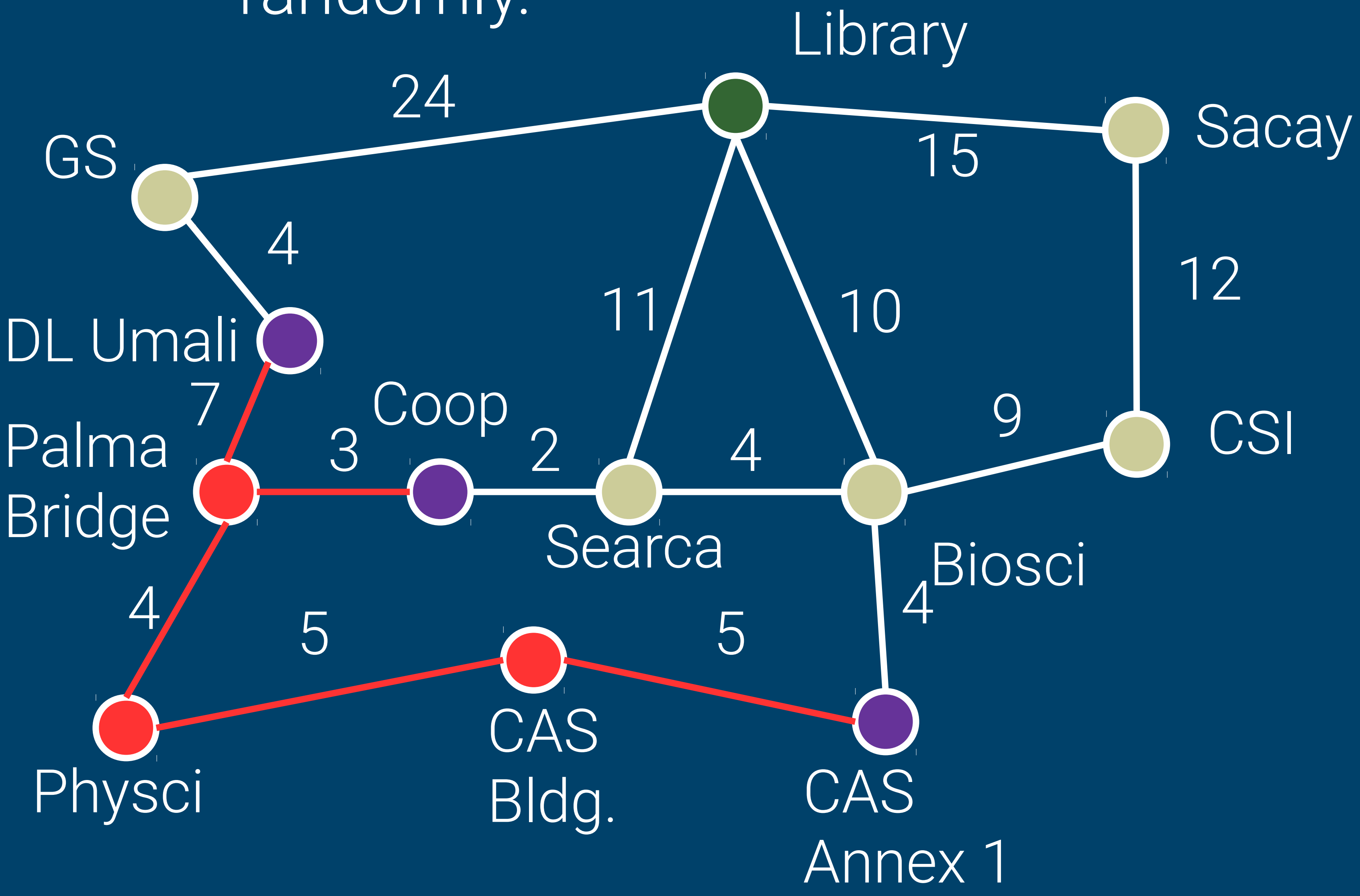


Keeping track of previously-visited nodes prevents backtracking.

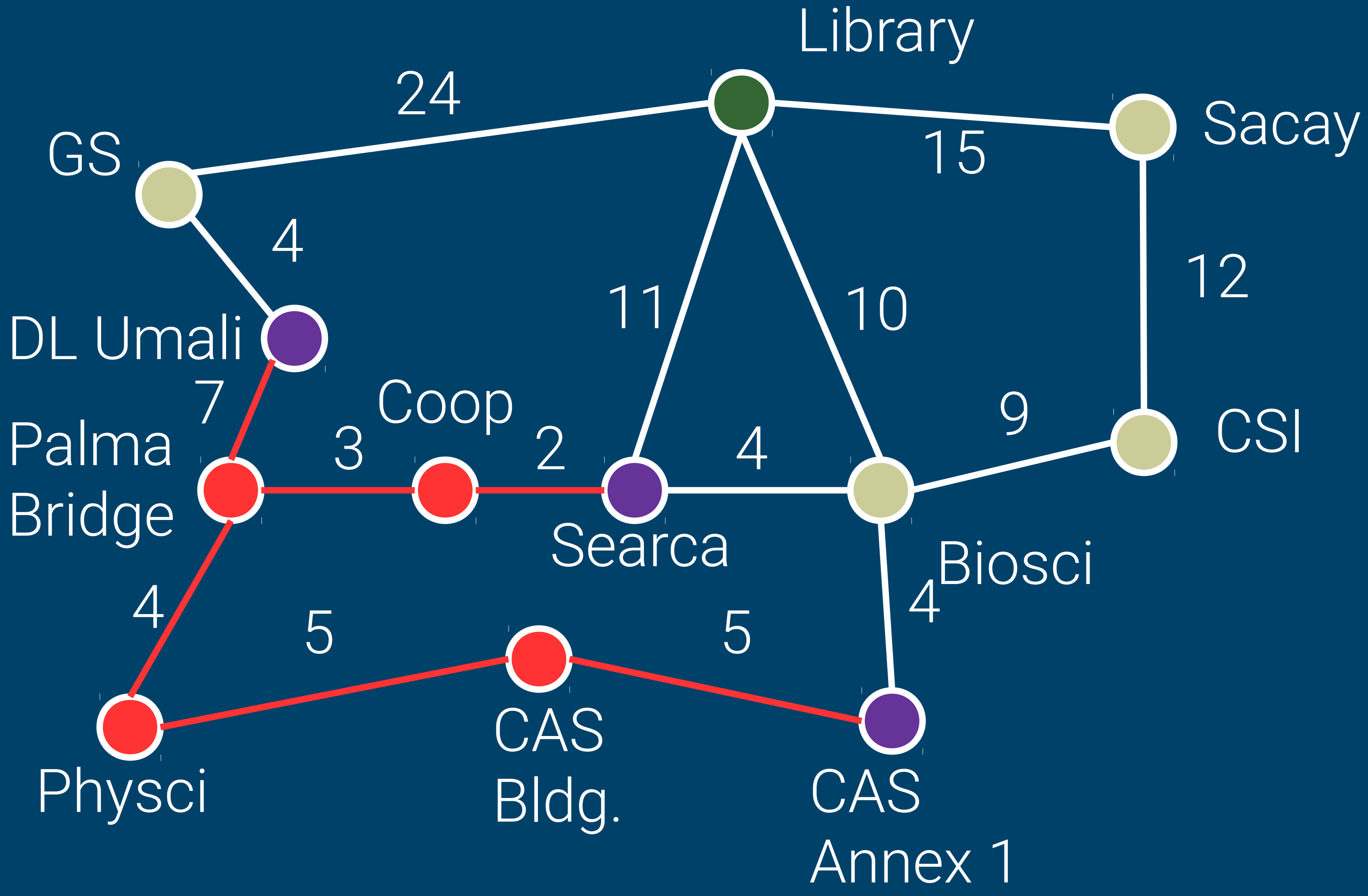


...and so on...

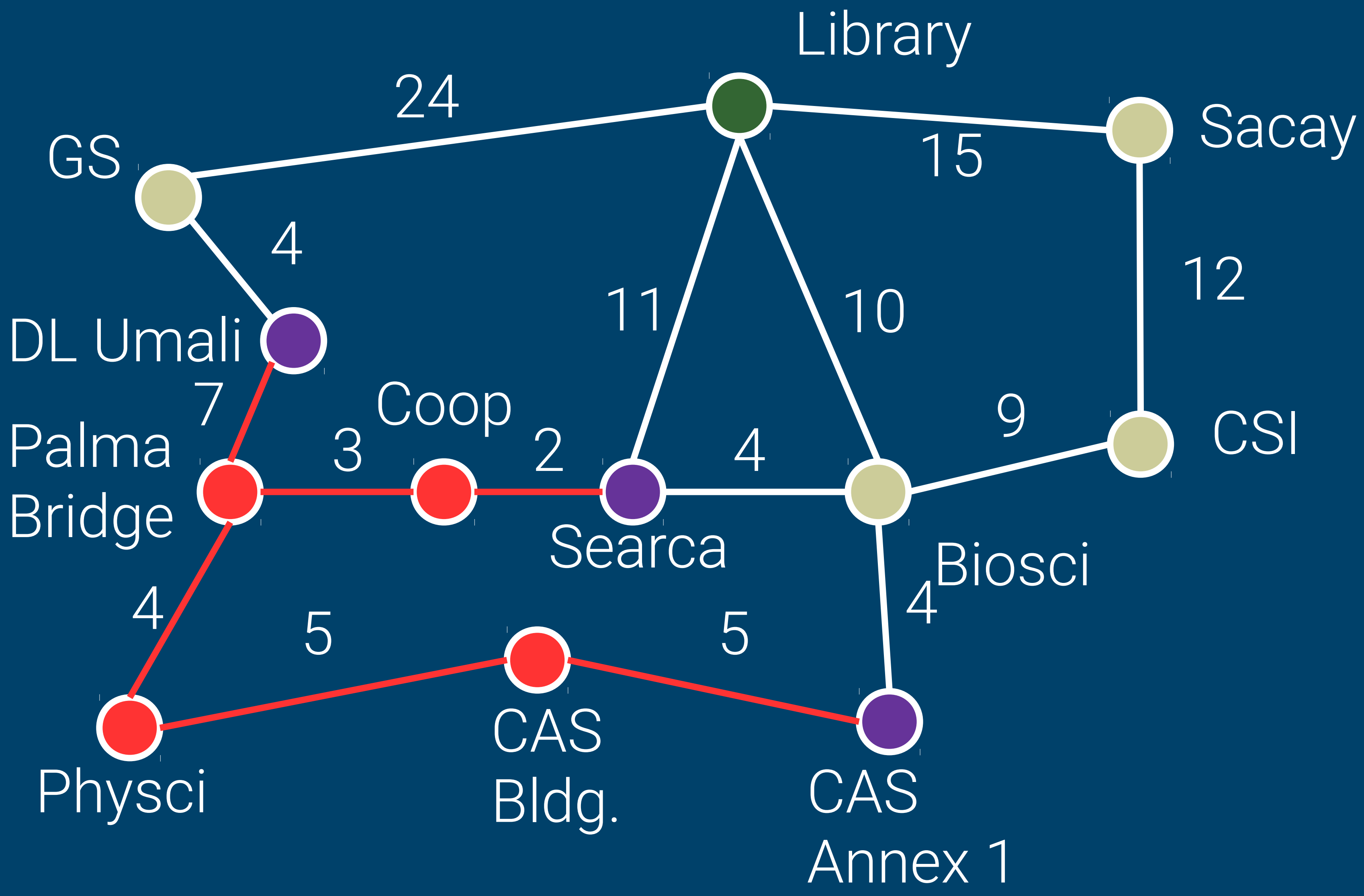
EXAMPLE. All paths are of length 2. Break ties randomly.



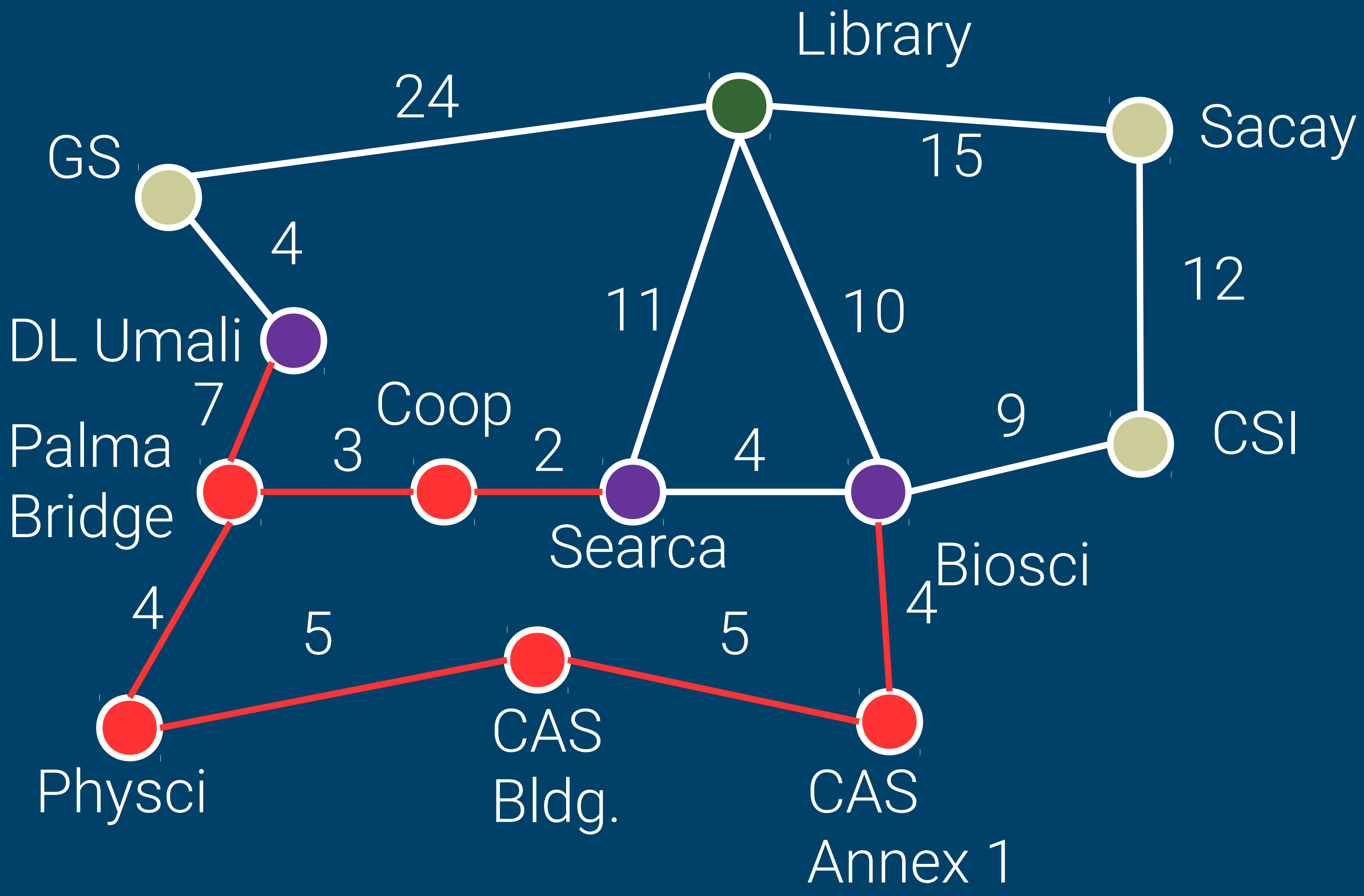
EXAMPLE. Say, “random,” tells us to expand the path from Coop...



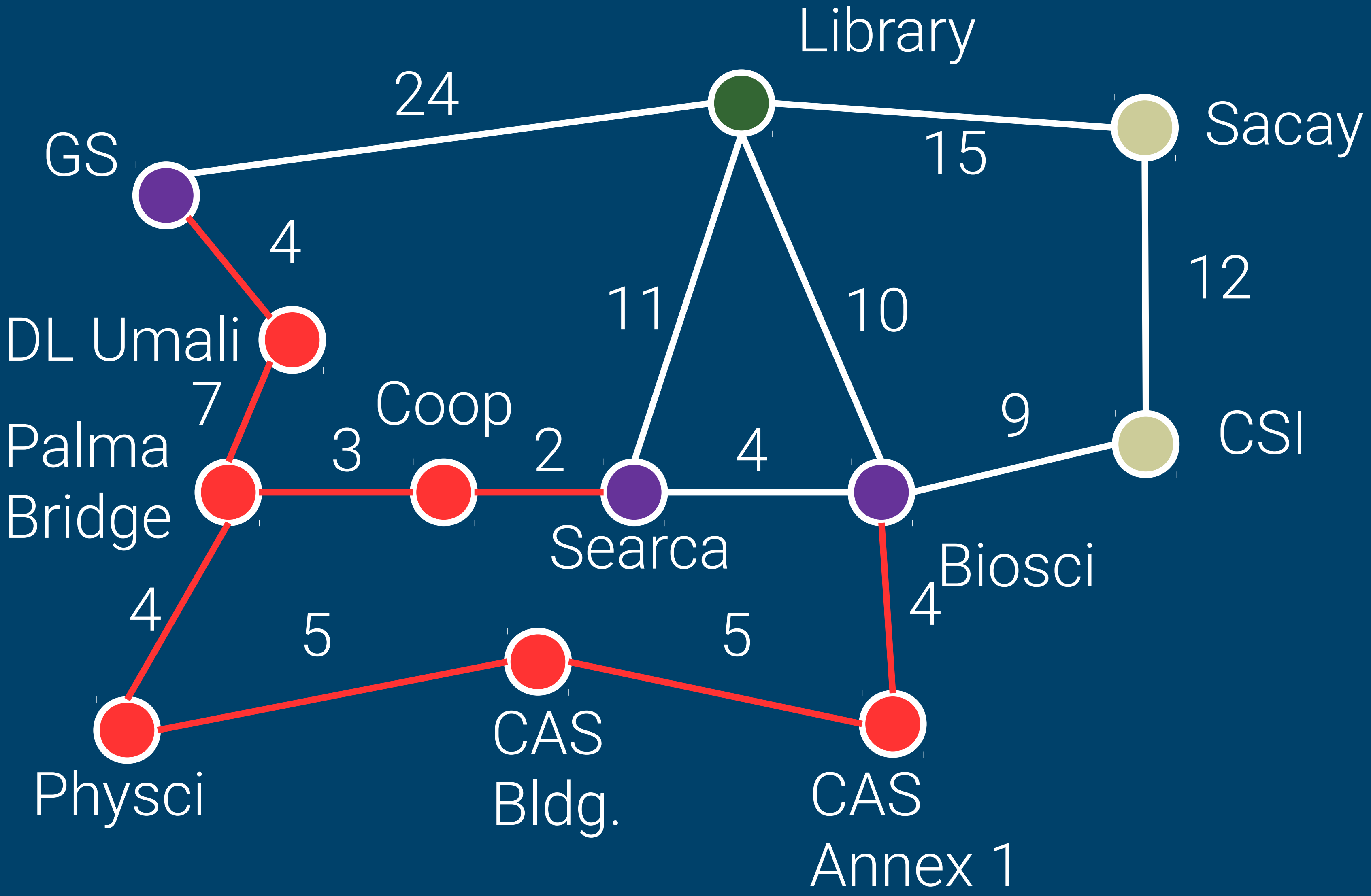
EXAMPLE. We now have two paths of length 2 and one path of length 3.



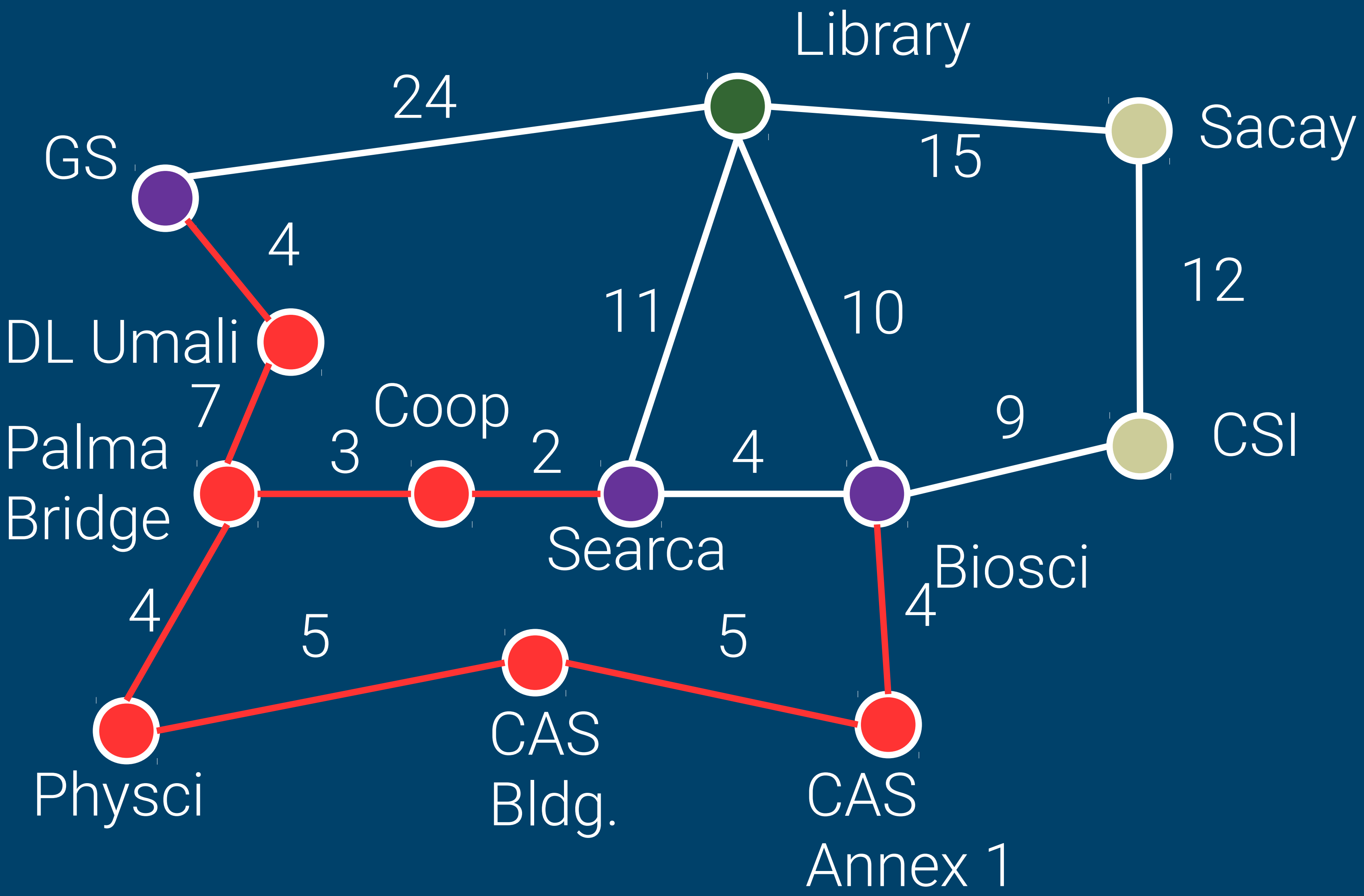
EXAMPLE. Expand one of the length 2 paths (in this case, CAS Annex 1).



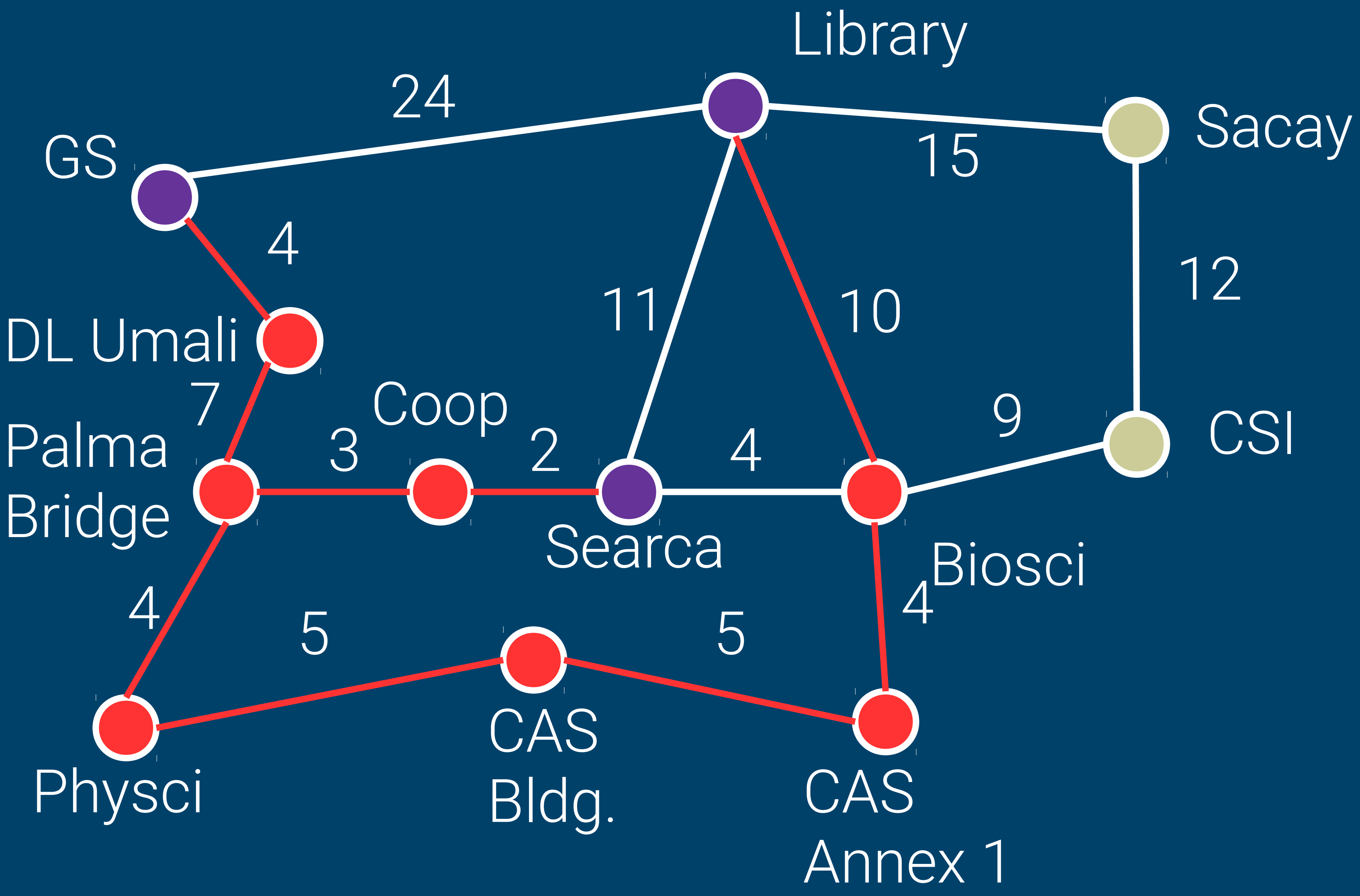
EXAMPLE. Expand the remaining length 2 path.



EXAMPLE. We now have three paths of length 3. Expand one of them...



EXAMPLE. We now have three paths of length 3. Expand one of them...

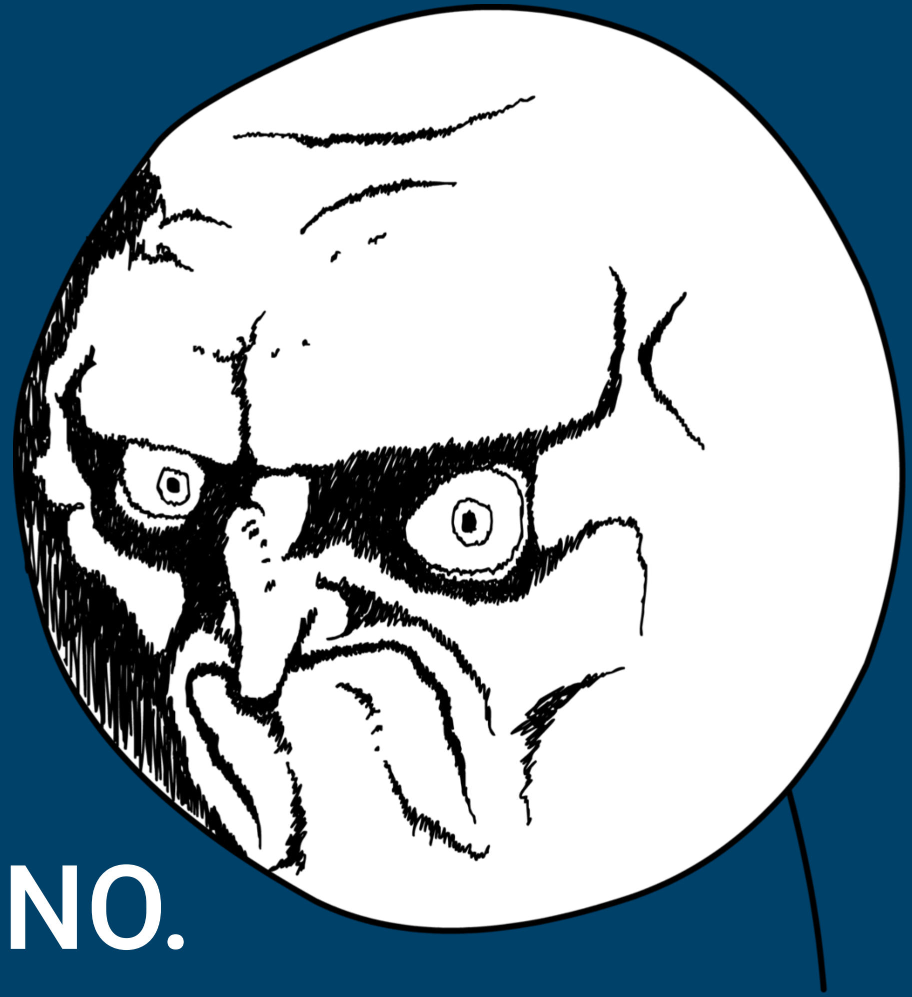


QUIZ (1/4)

1. Will we terminate our graph search algorithm when we add Library to the frontier (make it purple)?

ANSWER

The **goal test is applied** when we **remove a path from the frontier**, NOT when we add one.



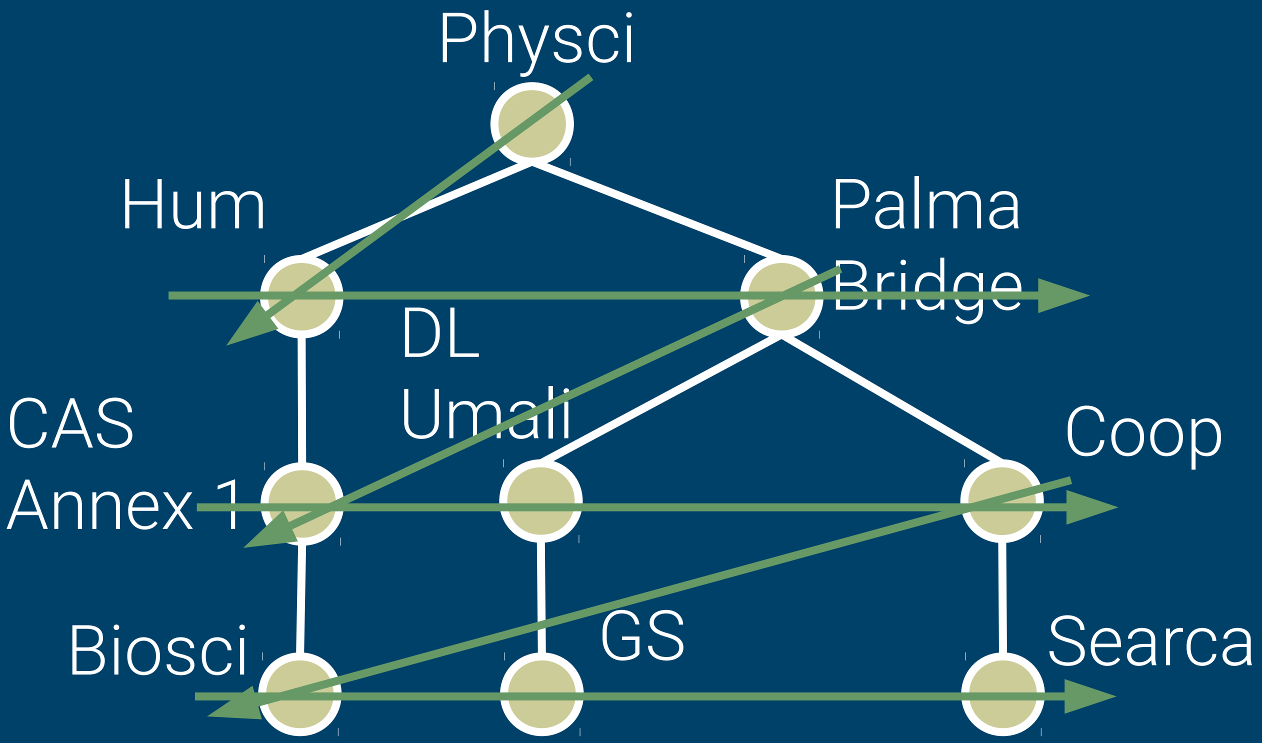
```
function graphSearch(problem) {
  frontier={[initial]}  explored={}
  while(frontier is not empty) {
    path=removeChoice(frontier)
    s=path.end  explored.add(s)
    if(GoalTest(s)) return path
    for (a in Actions(s)) //expand path
      if(Result(s,a) ∉ frontier ⋃ explored)
      frontier.add(path + s → Result(s, a))
  }
}
```

EXCEPTION

If we are **sure** that we are **getting the optimal path** upon **encountering the goal in the frontier**, we can modify graph search to **apply the goal test upon adding a path to the frontier**.

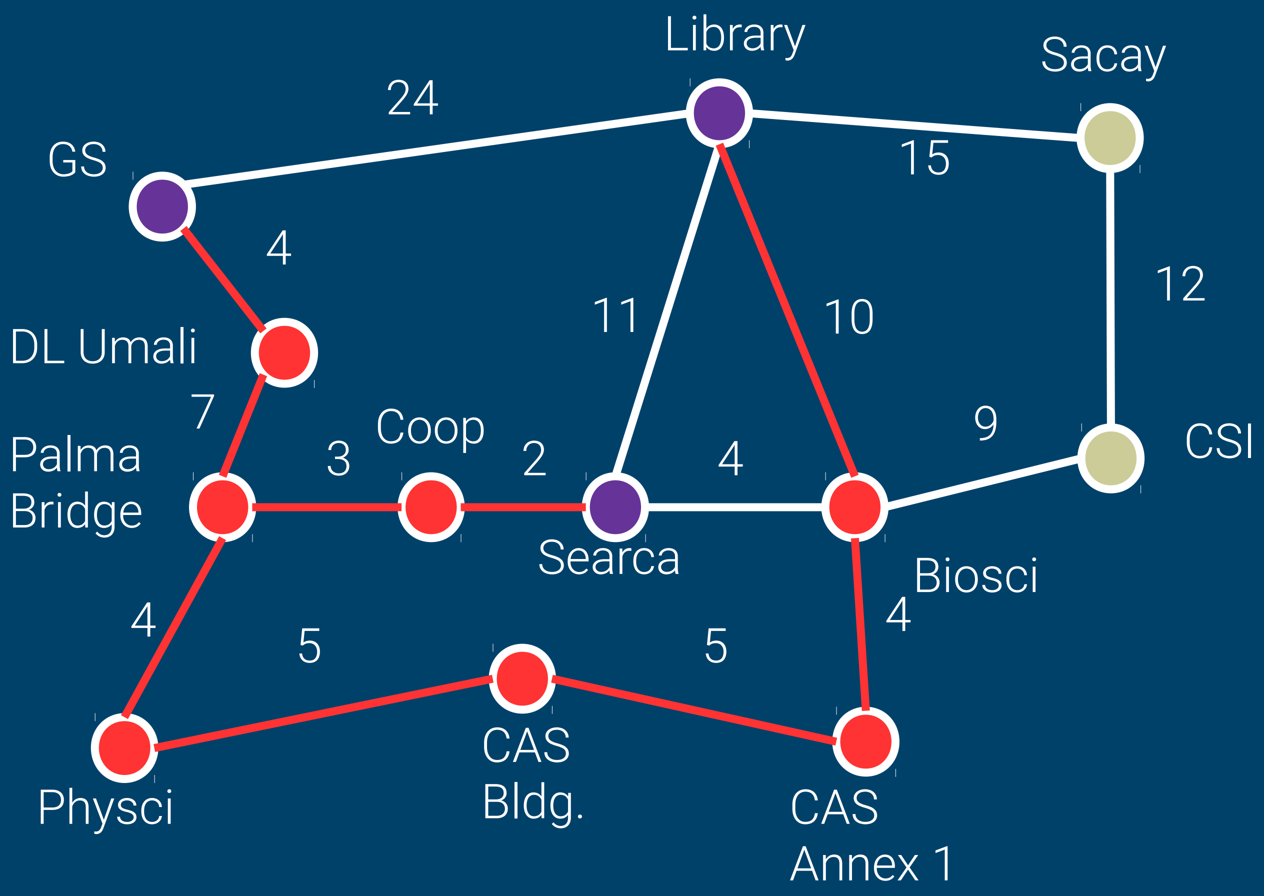
Why Breadth-First?

The **order** in which the state space is explored **expands the breadth of the tree first**.

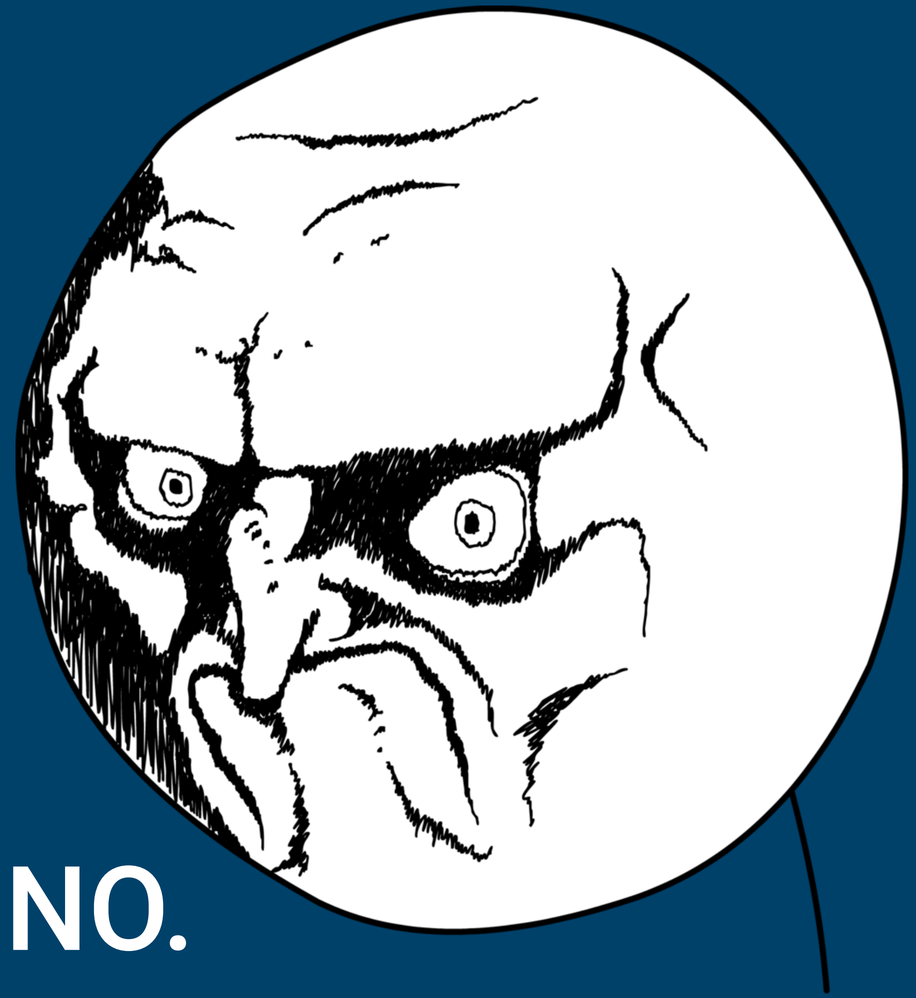


QUIZ (CONT'D)

2. Is the path we found the best/optimal path, based on cost?



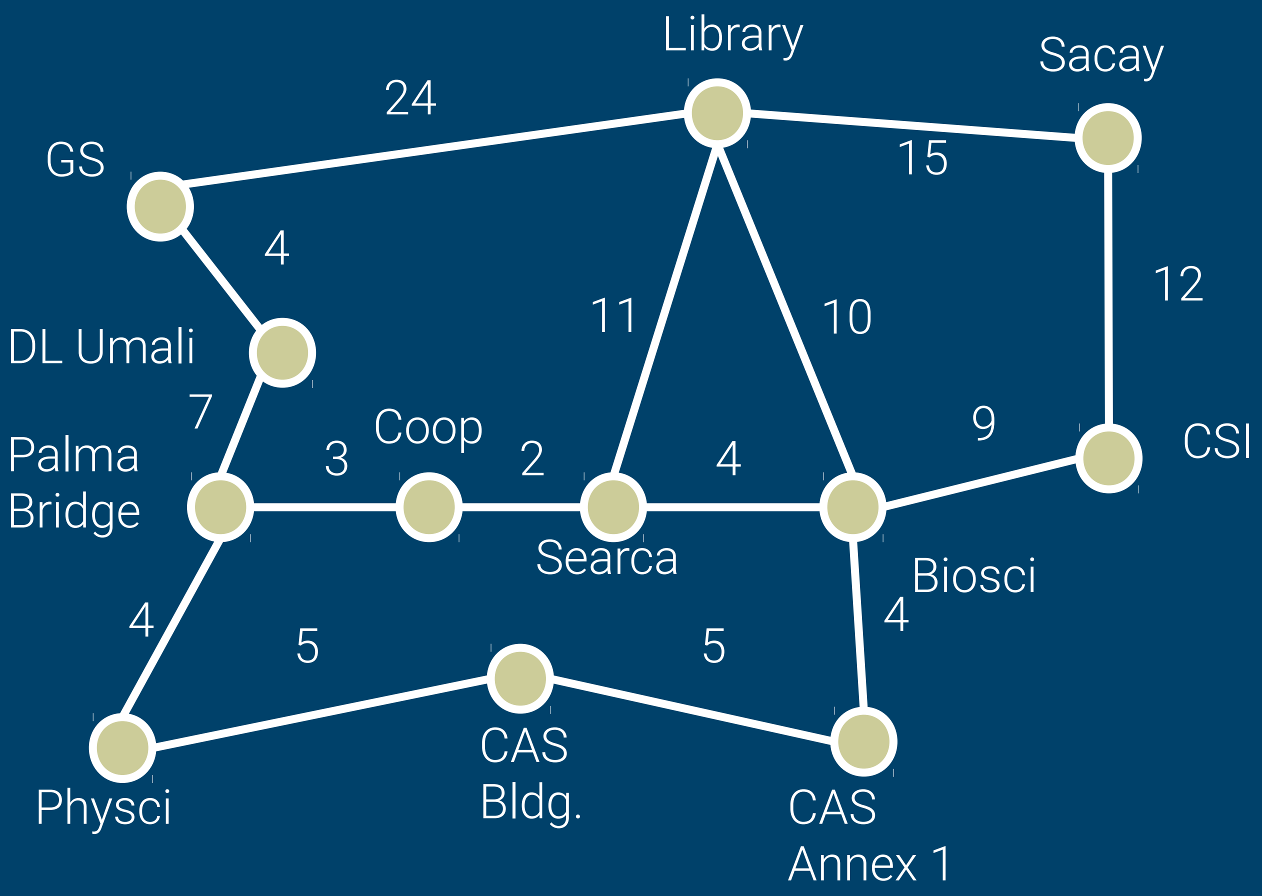
ANSWER



This version of **breadth-first search** will **always** find the **shortest path** in terms of **length/number of steps**, but **NOT cost**,and is **optimal** if and only if **path length = path cost**.

QUIZ (CONT'D)

3. Identify the optimal path in terms of cost, and;
4. specify its cost.



ANSWER

Physci → Palma Bridge → Coop → Searca → Library

Cost = 20

QUIZ (CONT'D)

5 – 8. Identify the properties of the environment of a robot car agent.

9. What process is the continuous loop that occurs when AI agents sense and interact with its environment?

10. Identify a reason for uncertainty.

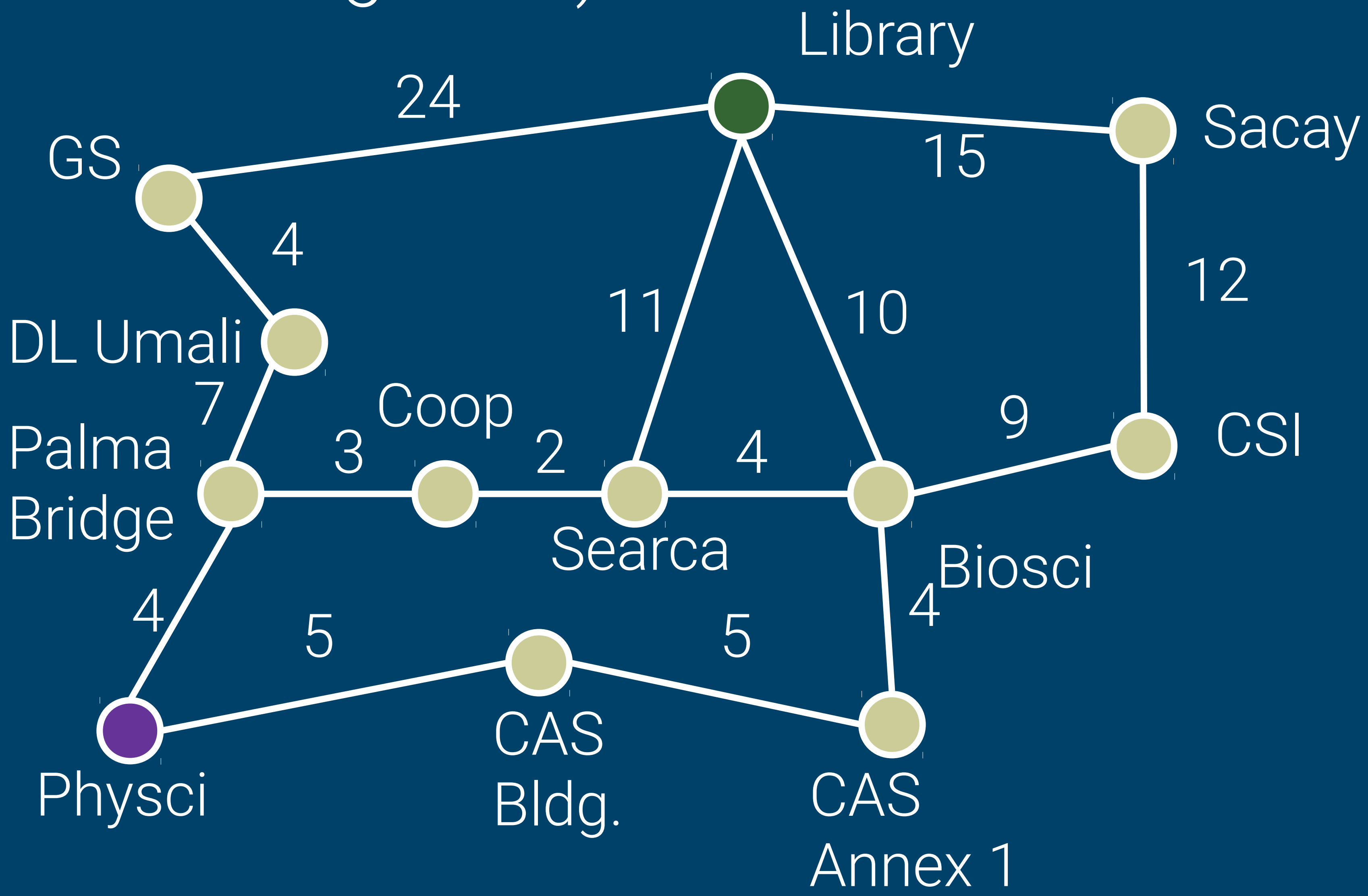
Uniform-Cost Search

An algorithm that chooses the **path** with the **cheapest cost** to be **expanded**.

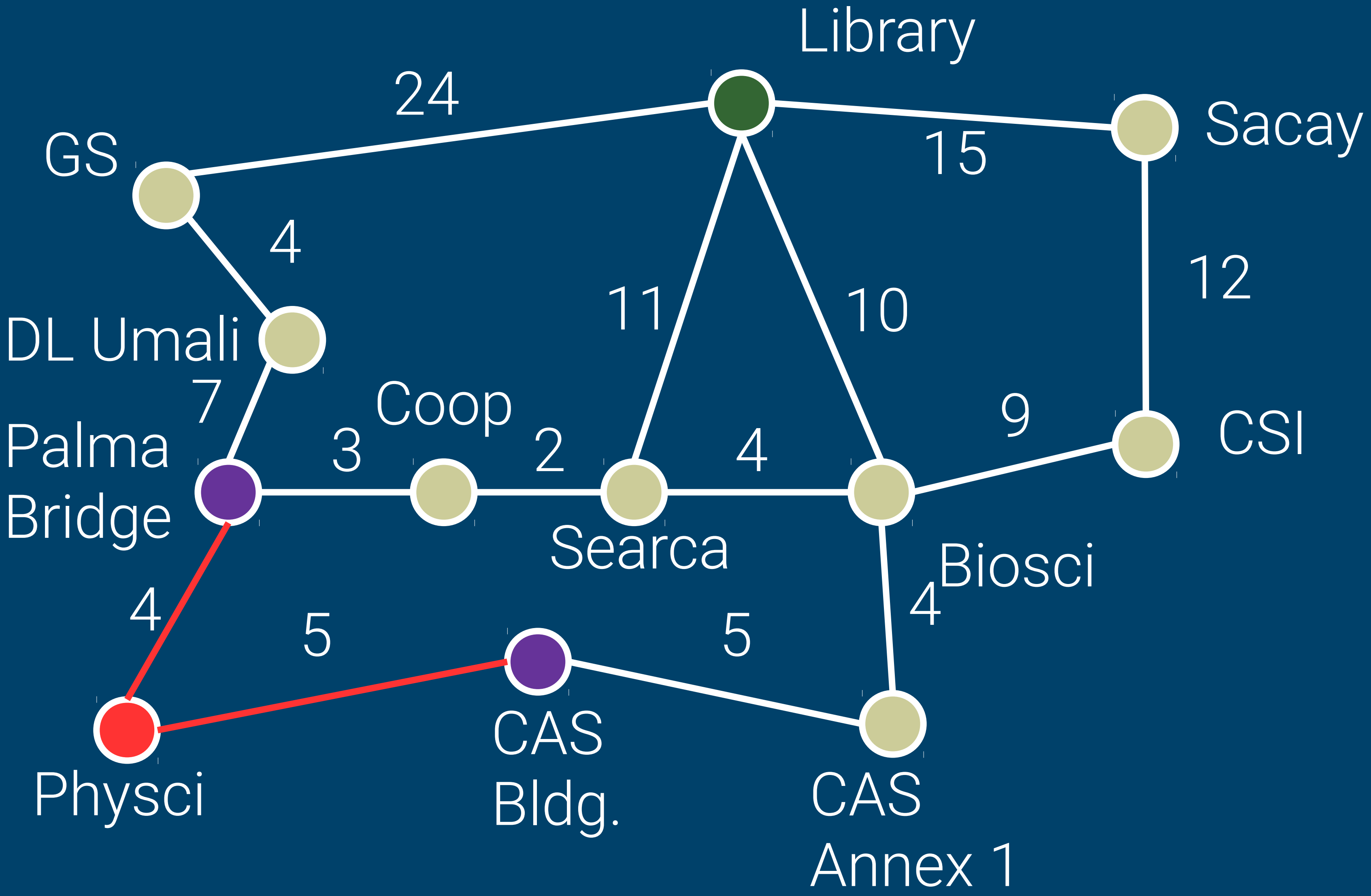
However, in order to **ensure** that the path it finds is the **cheapest**, **all paths** leading **to the goal** must be **explored**.

```
function graphSearch(problem) {
  frontier={[initial]}  explored={}
  while(frontier is not empty) {
    path=removeChoice(frontier)
    s=path.end  explored.add(s)
    if(GoalTest(s)) return path
    for (a in Actions(s)) //expand path
      if((Result(s,a) $\notin$ frontier $\bigcup$ explored) or
         GoalTest(Result(s,a))
      frontier.add(path + s → Result(s, a))
  }
}
```

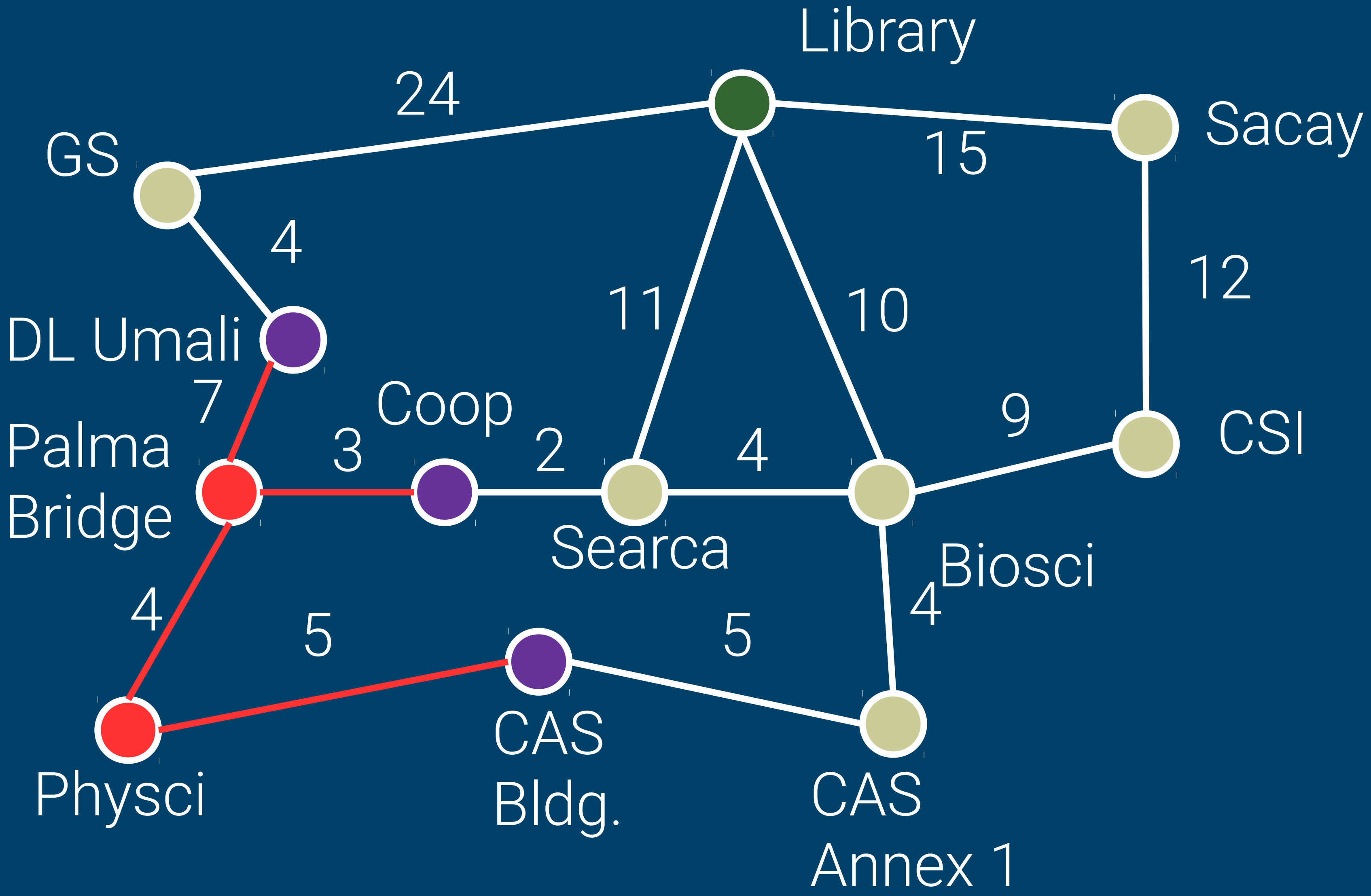
EXAMPLE. Add the initial state first (path length = 0).



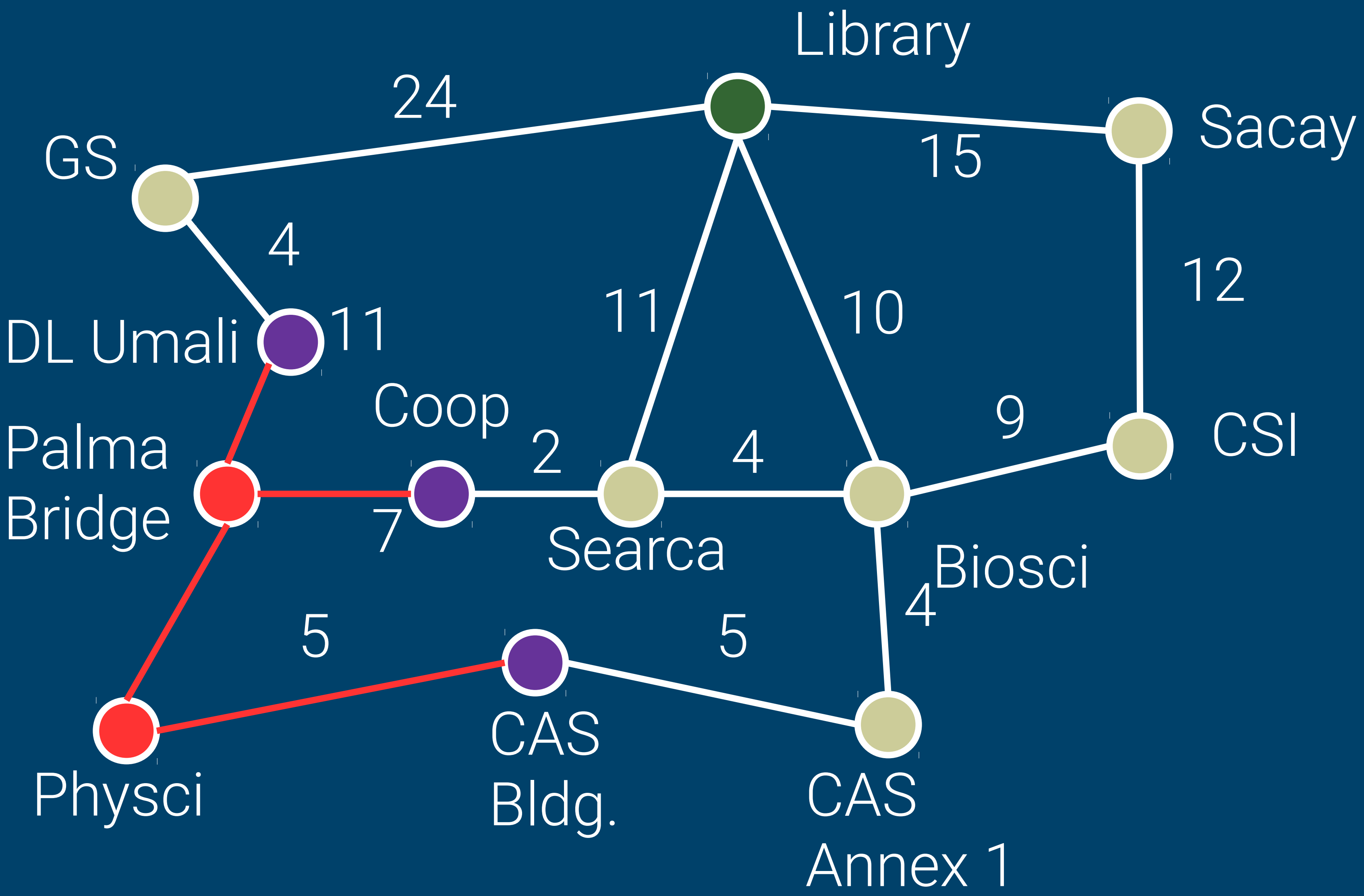
EXAMPLE. Remove the initial state from the frontier and add its successors.



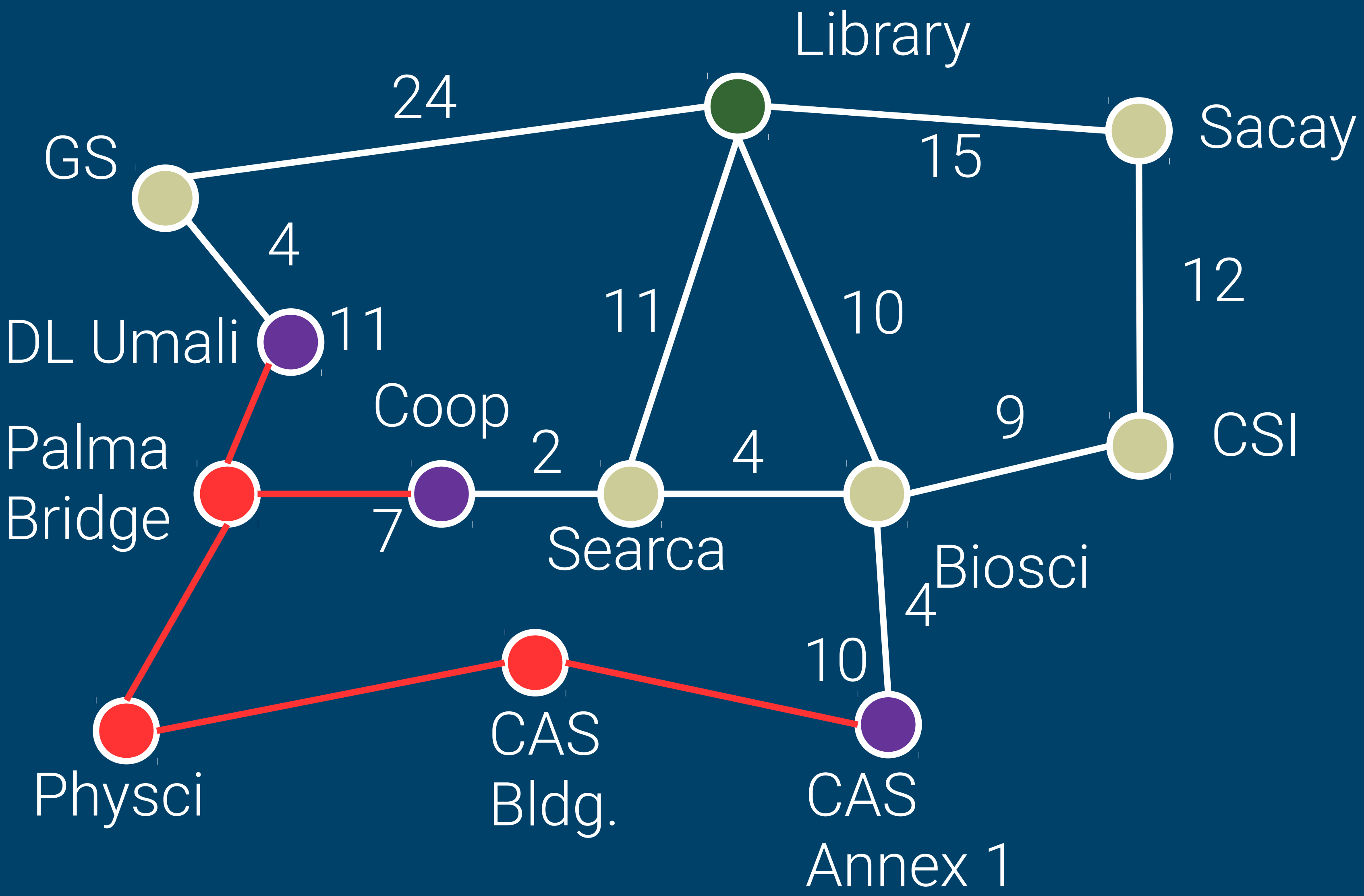
EXAMPLE. There are two possible paths; expand the cheaper one.



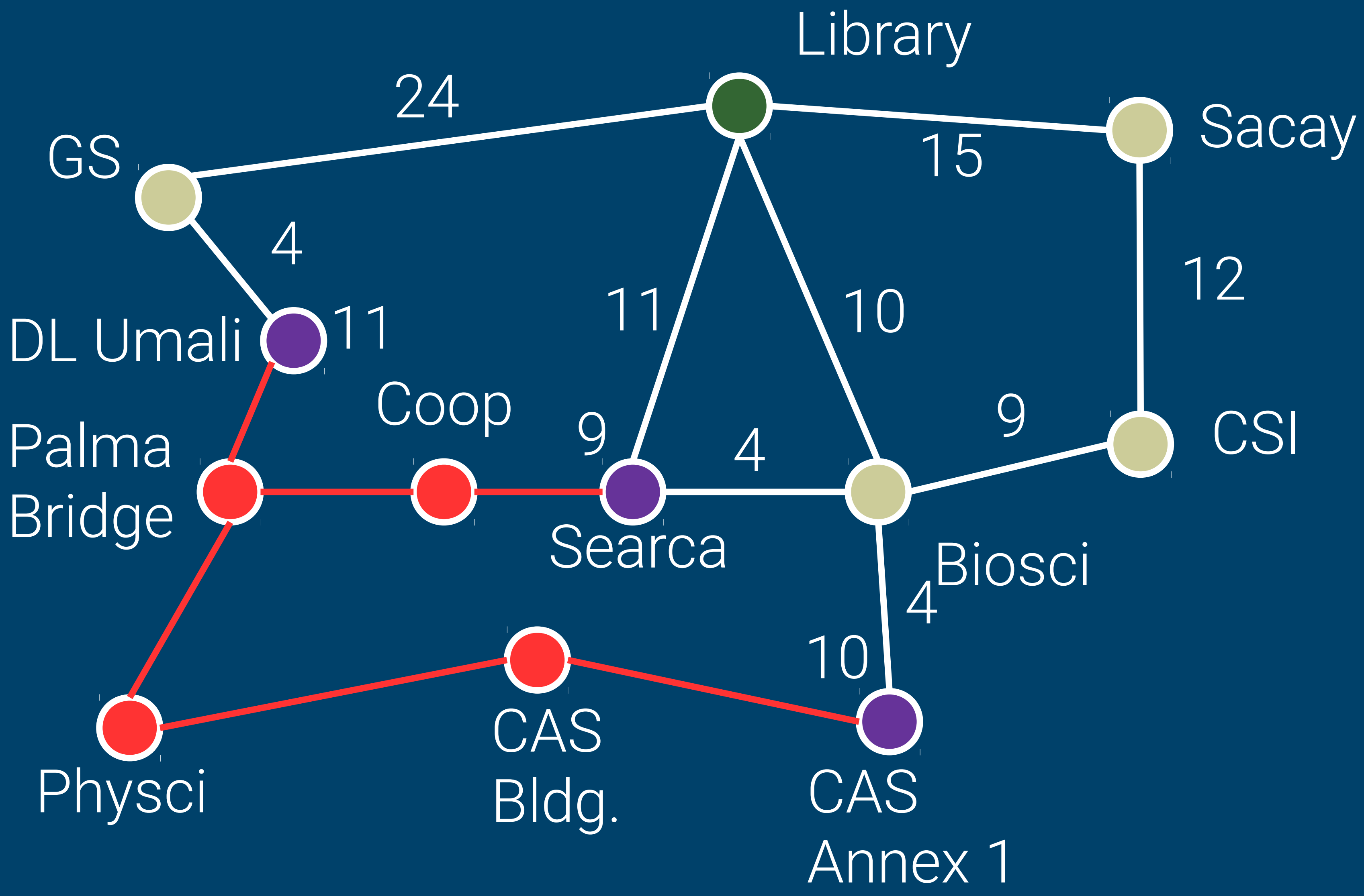
EXAMPLE. Recompute the new path costs.



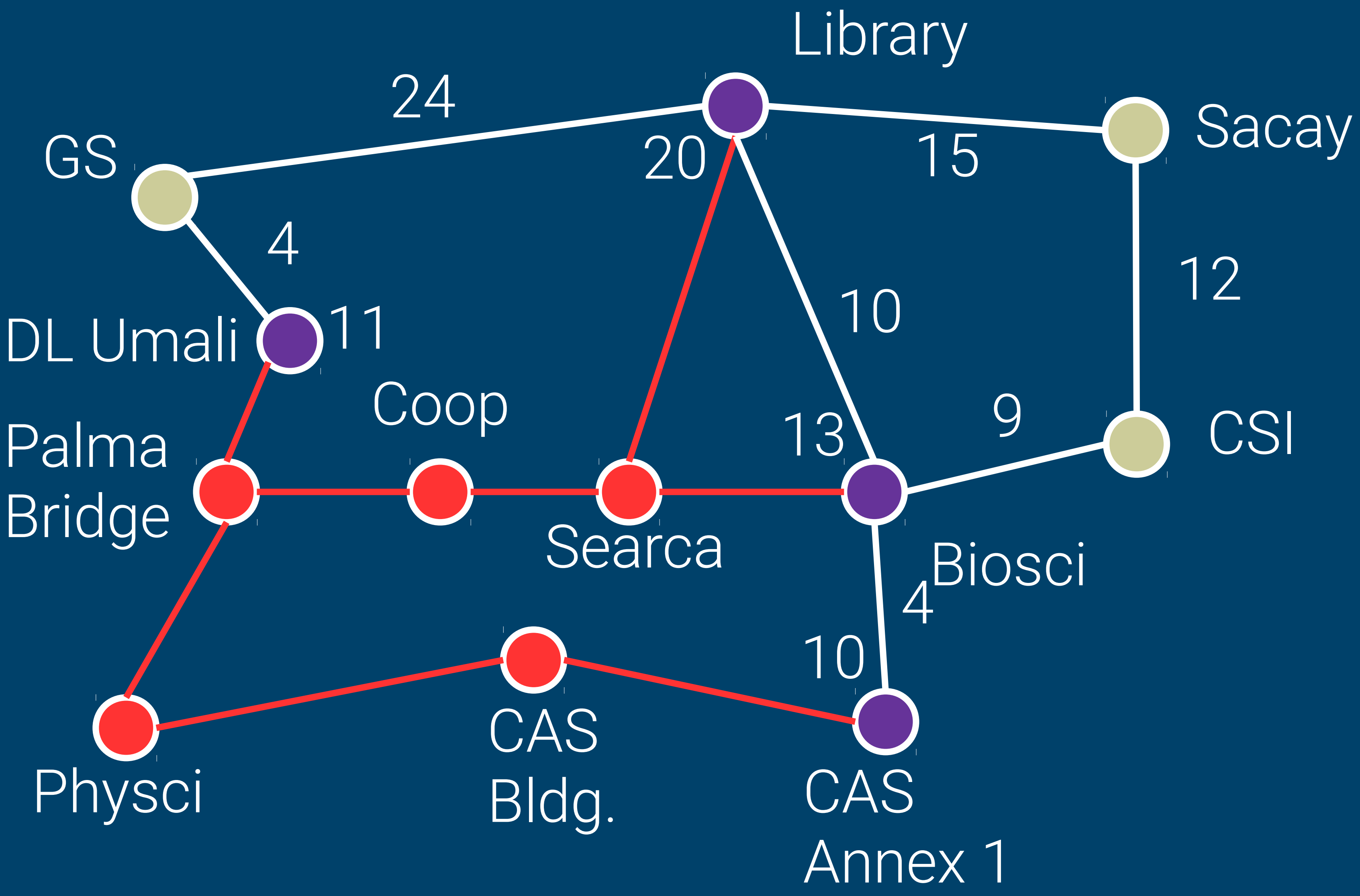
EXAMPLE. Choose the cheapest path and recompute its cost.



EXAMPLE. Choose the cheapest path and recompute its cost.



EXAMPLE. Choose the cheapest path and recompute its cost.

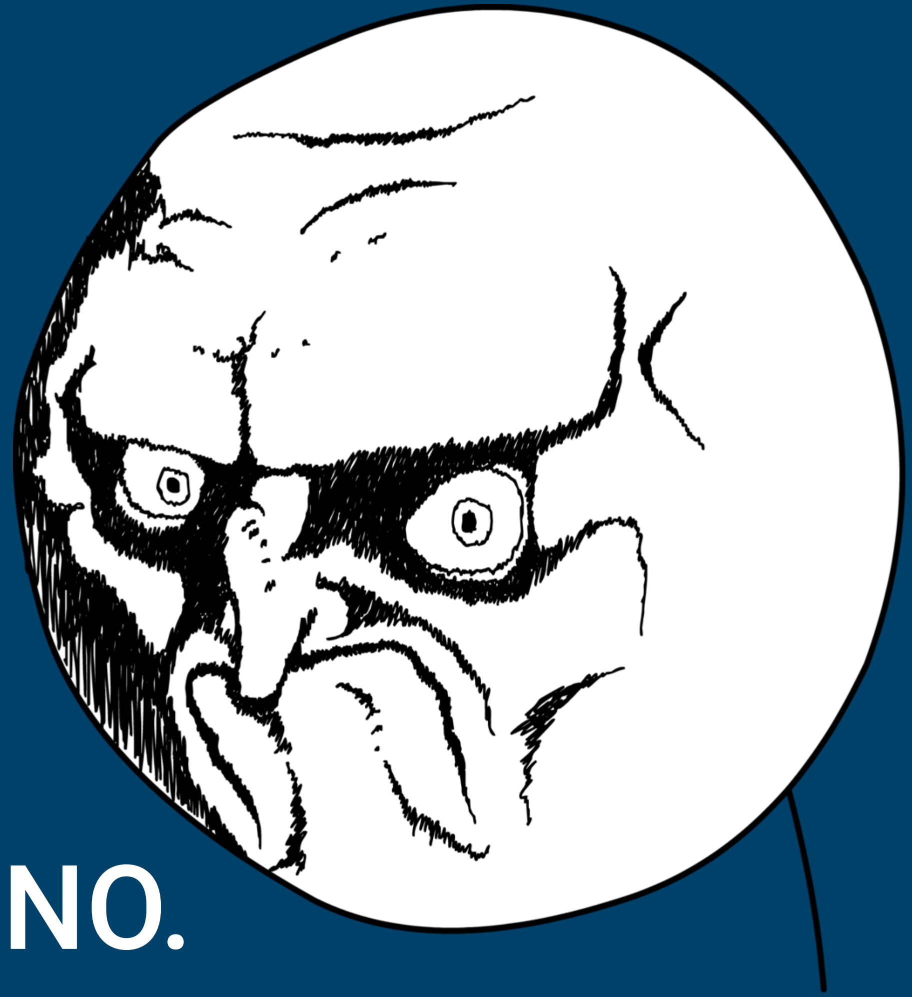


QUIZ (1/4)

1. Do we stop upon expanding to Library (Yes/No)?

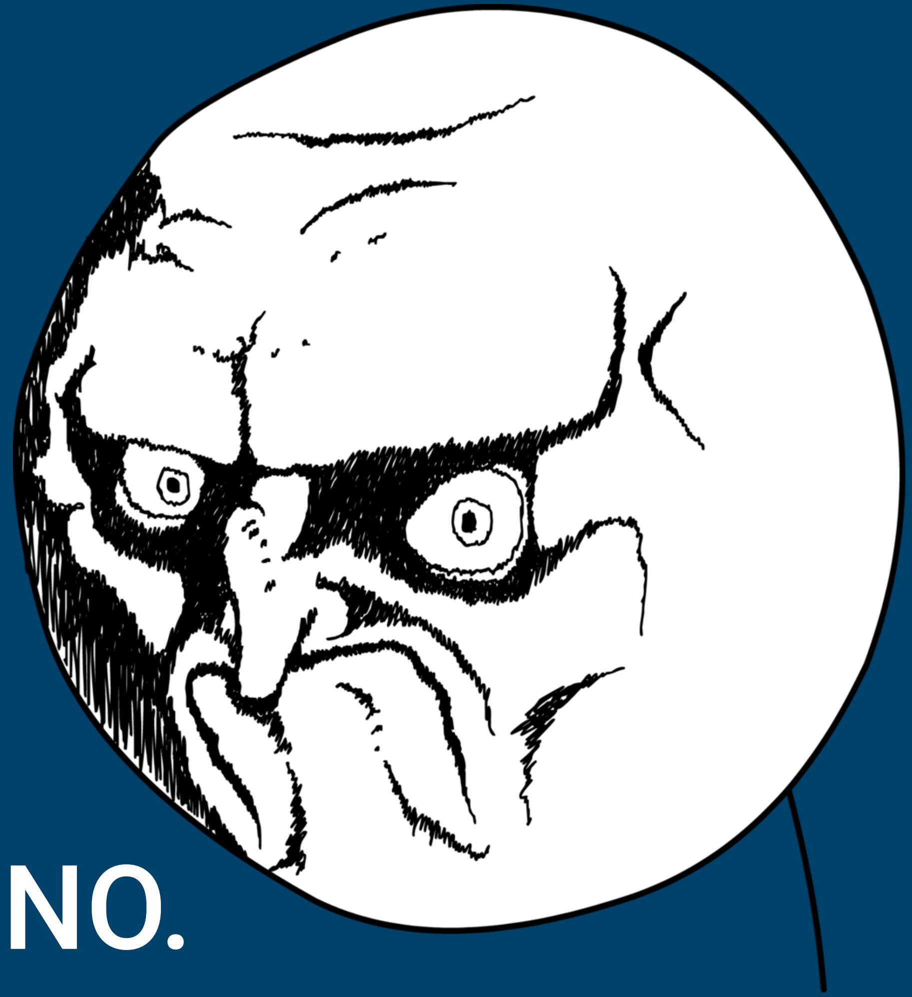
ANSWER

Again, the **goal test is applied** when we **remove a path from the frontier**, NOT when we add one.

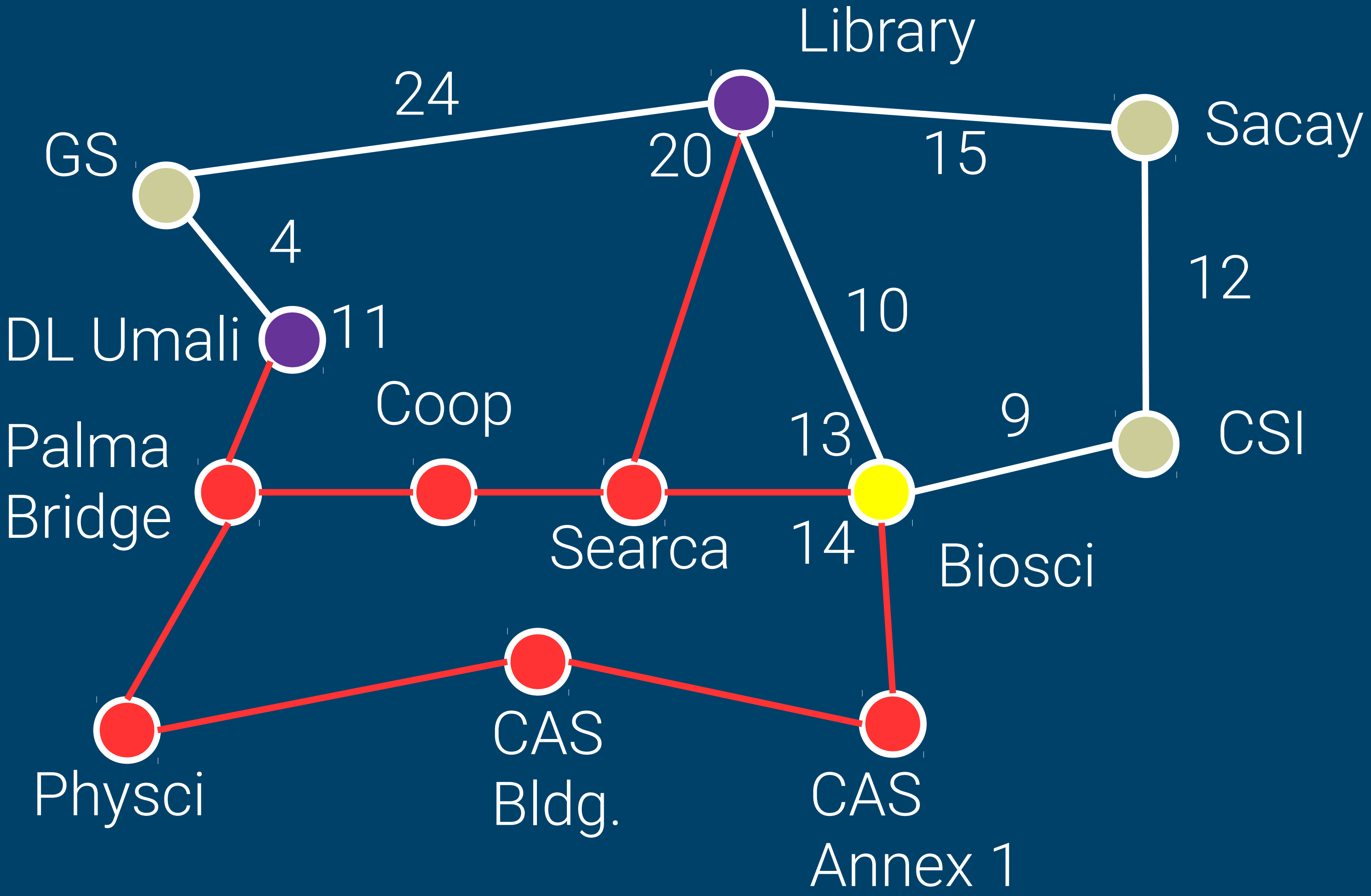


ANSWER

Moreover, we need to **compare this path** with **all other paths** from the initial state to the goal.



EXAMPLE. Choose the cheapest path and recompute its cost.



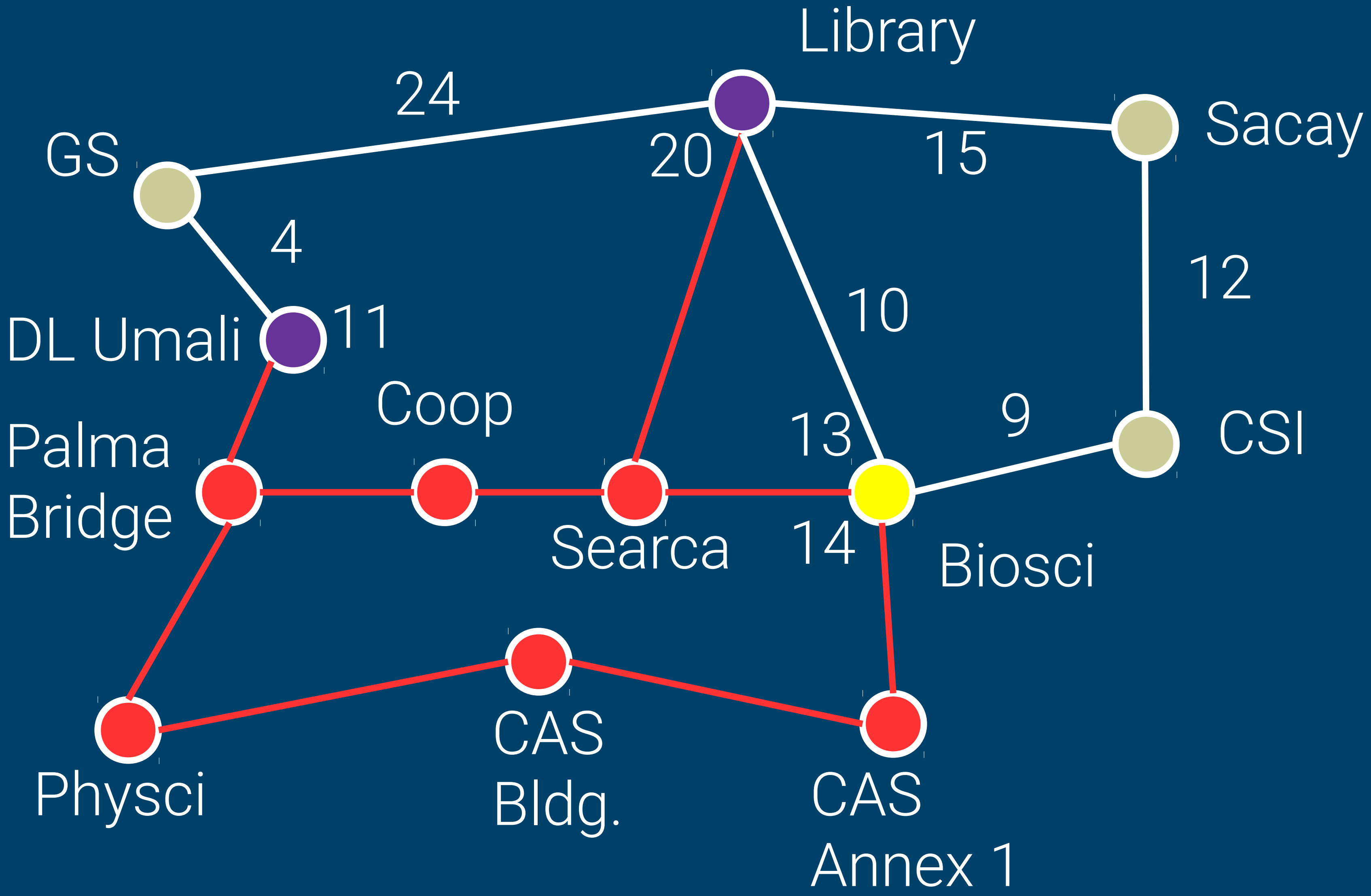
QUIZ (1/4)

2. Do we add Biosci to the frontier (Yes/No)?

ANSWER

NO!!!

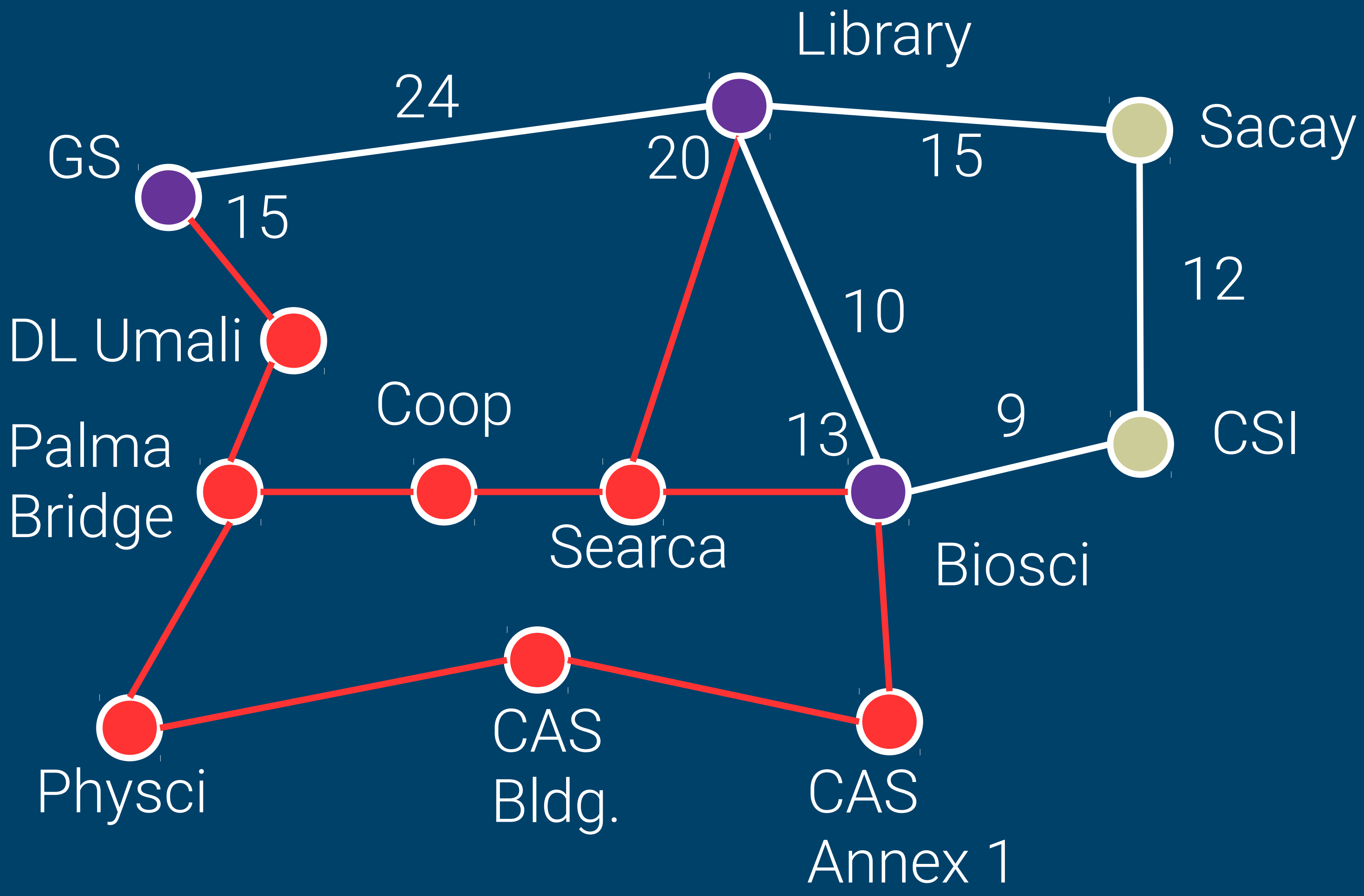
EXAMPLE. Biosci is already in the frontier, AND the new path is **more expensive** (14 > 13).



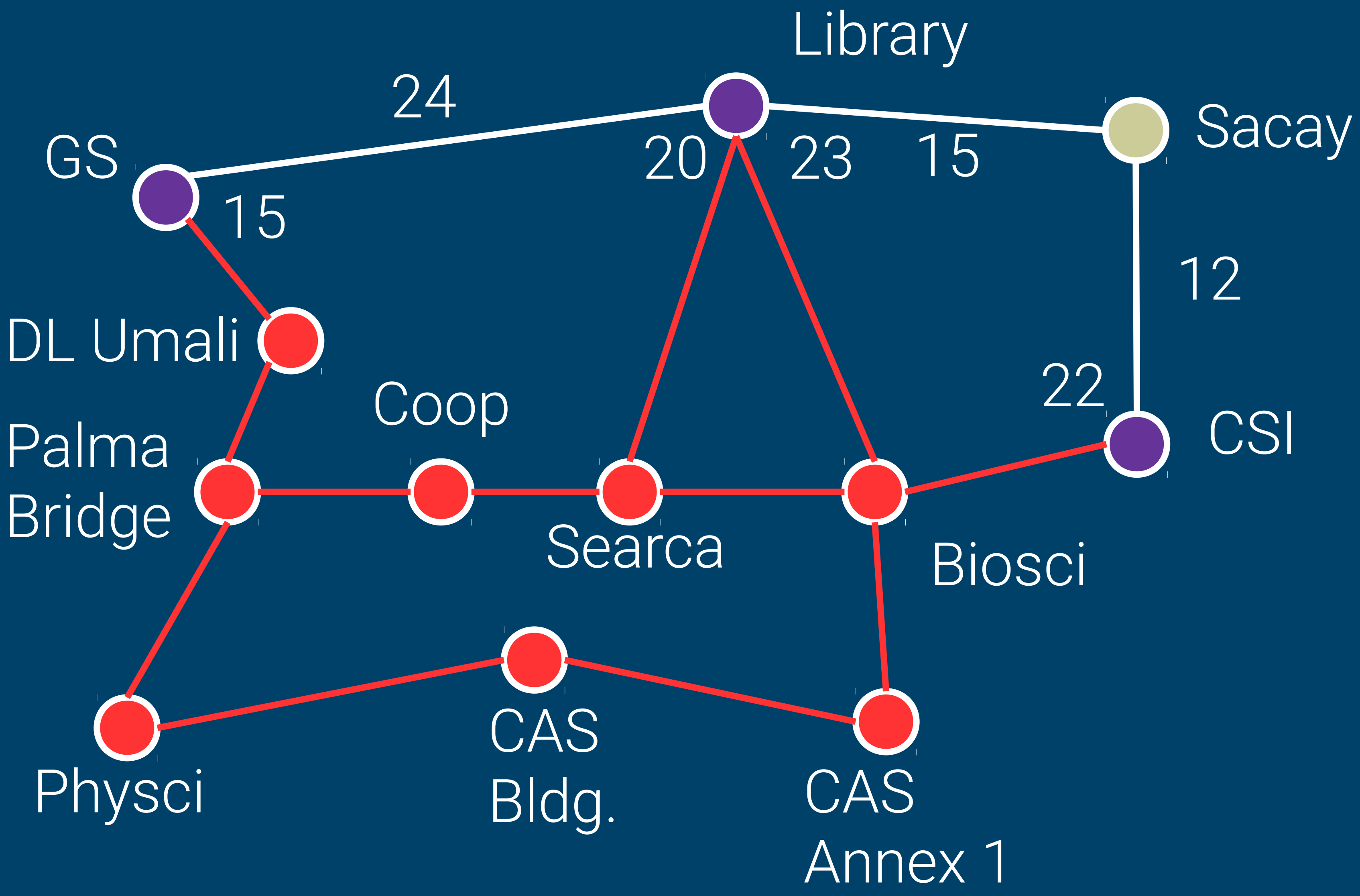
We can amend the condition for adding to the frontier to consider this possibility:

```
if((Result(s,a) $\notin$ frontier $\cup$ explored)
or GoalTest(Result(s,a))
or ((Result(s, a) $\in$ frontier $\cup$ explored) and PathCost(s) < PathCost(duplicate))
```

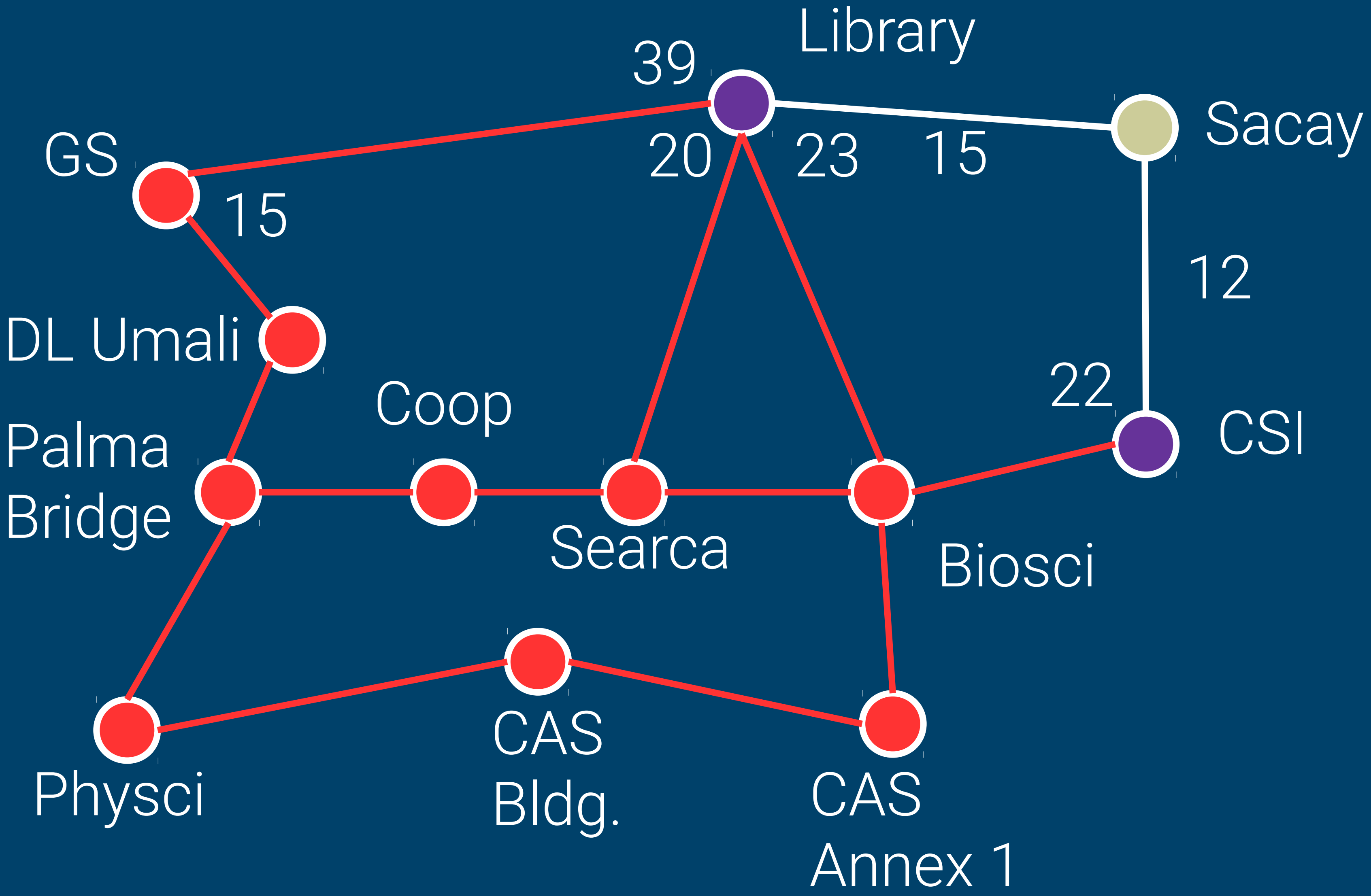
EXAMPLE. Choose the cheapest path and recompute its cost.



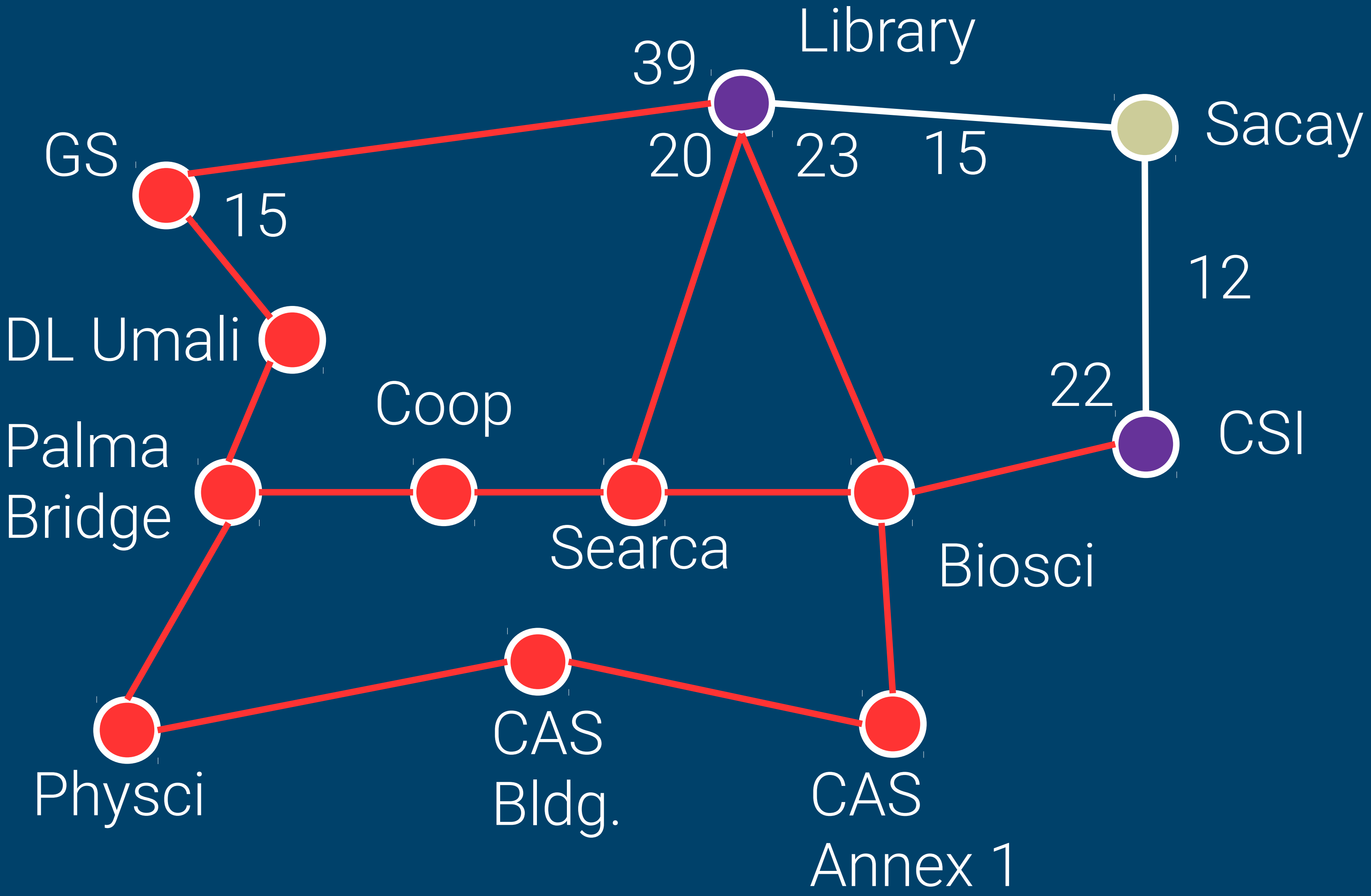
EXAMPLE. Choose the cheapest path and recompute its cost.



EXAMPLE. Choose the cheapest path and recompute its cost.



EXAMPLE. The next cheapest path is from Searca to Library (cost = 20).



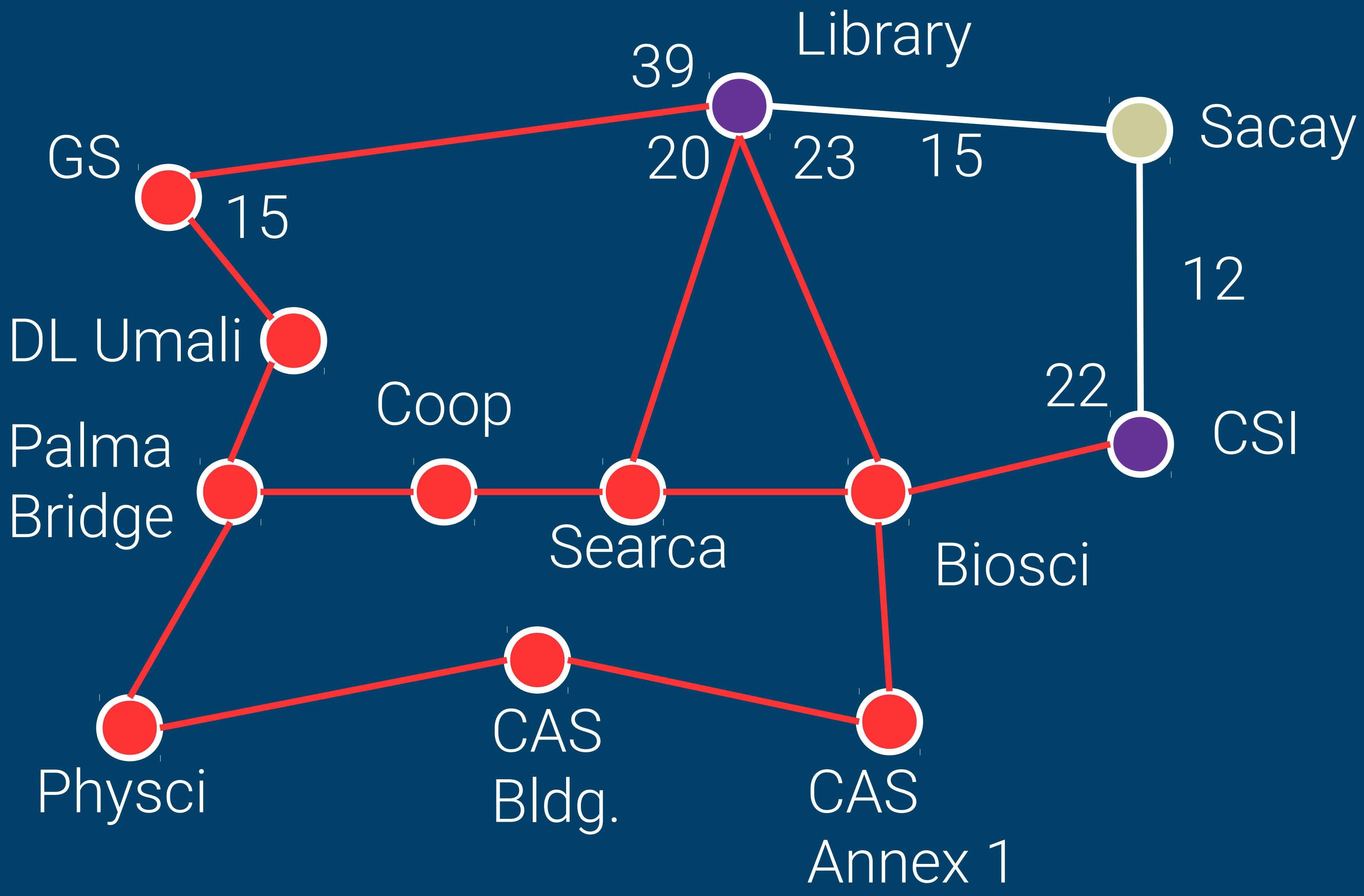
QUIZ (1/4)

3. Now do we stop (Yes/No)?

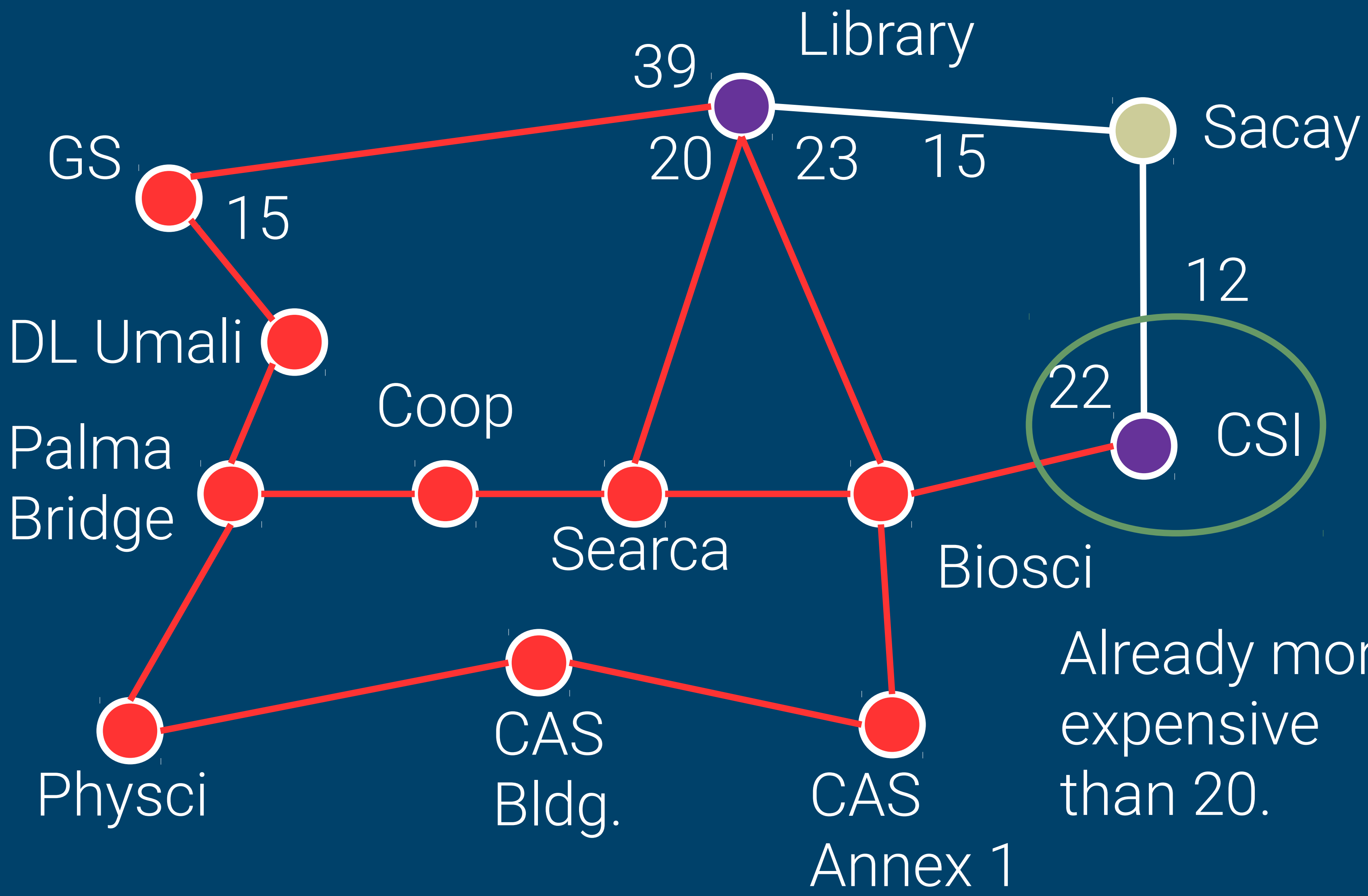
ANSWER

When Library is removed, the goal test is applied and we see that it is the goal.

EXAMPLE. Note that we pick the path of cost 20 over those of cost 23 and 39.

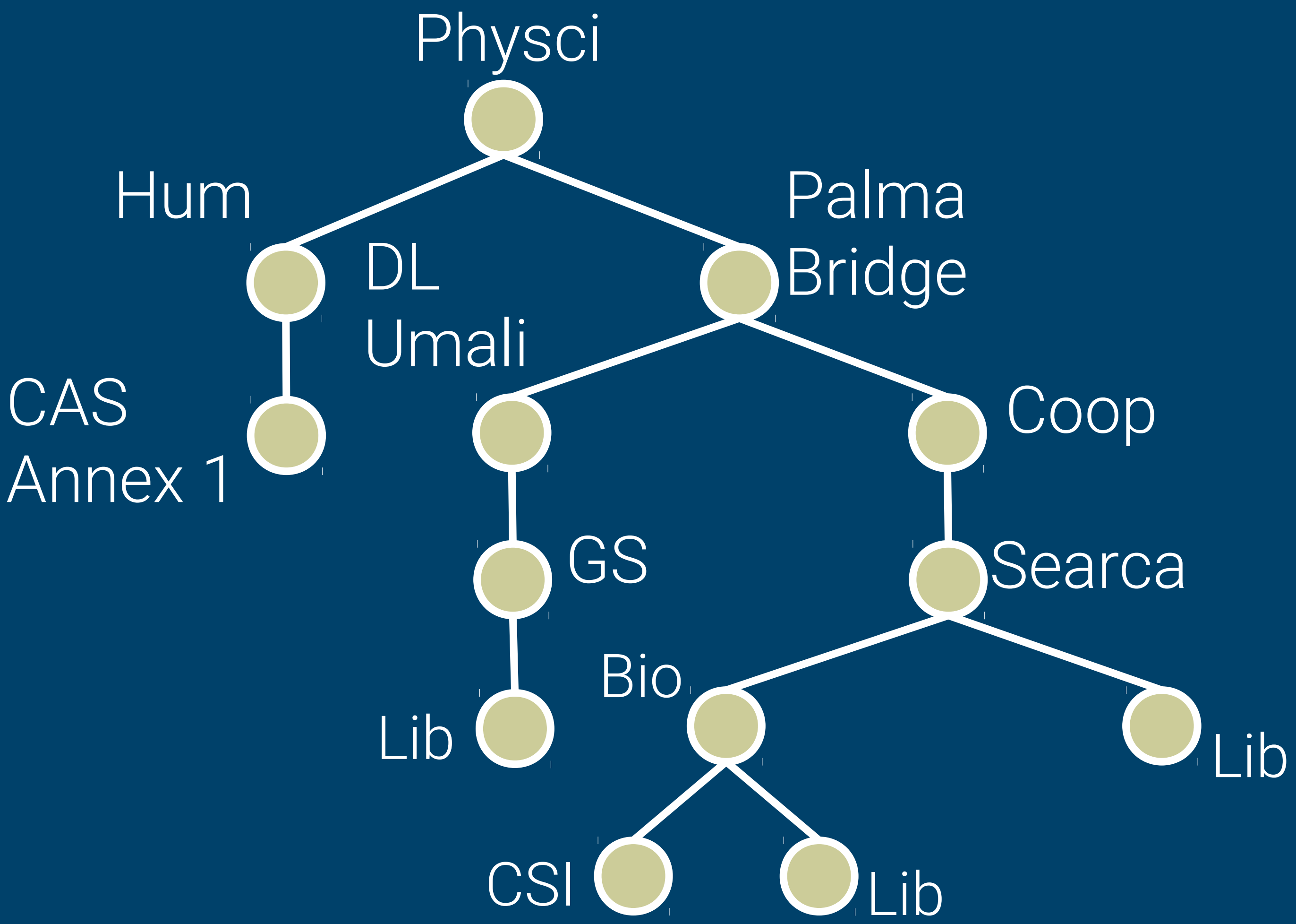


EXAMPLE. Moreover, the unfinished path of cost 22 is no longer expanded.



Already more expensive than 20.

The superimposed tree looks like:



Depth-First Search

Tree search algorithm that chooses the **path with the longest length** (again, not cost) among unexplored paths in the frontier, thus prioritizing **depth over breadth**.

Optimal Algorithm

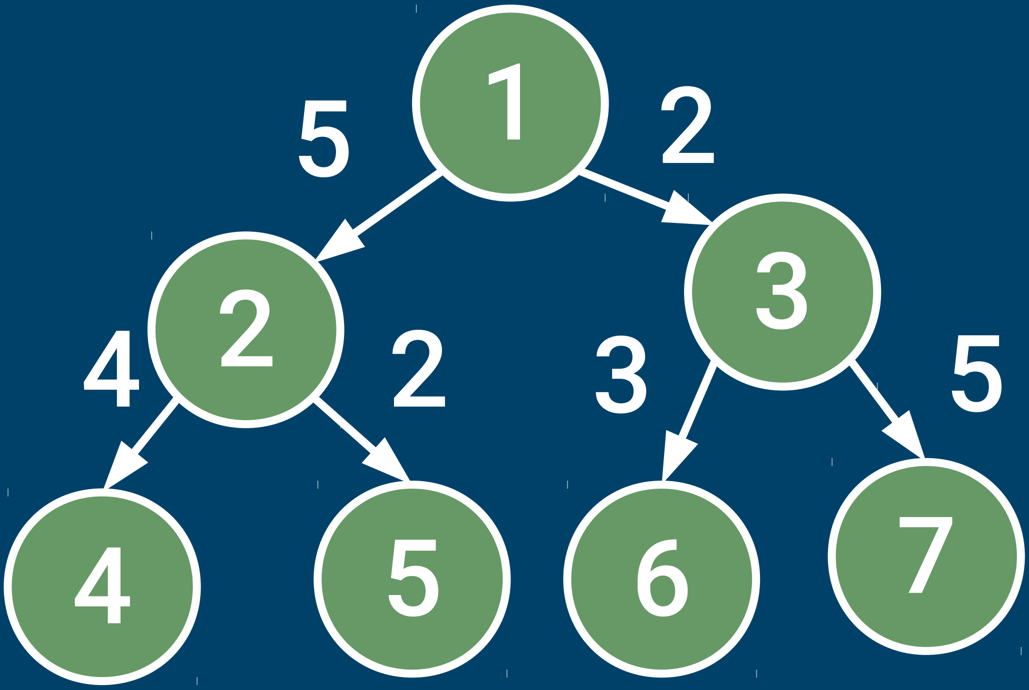
A search algorithm that is **guaranteed** to find the **optimal** (best) **solution**.

Complete Algorithm

A search algorithm that is **guaranteed** to **find the goal**, even though it **might not be optimal**.

QUIZ (1/4)

1. In what order will the following tree be traversed using...
   a. Breadth-first search
   b. Depth-first search
   c. Uniform-cost search



Example order:
1 7 3 6 5 2 4

QUIZ (CONT.)

2. If we are looking for the **shortest path**, which of the following is/are **optimal**?
    a. Breadth-first search
    b. Depth-first search
    c. Uniform-cost search

QUIZ (CONT.)

3. If we are looking for the **cheapest path**, which of the following is/are **optimal**?
   a. Breadth-first search
   b. Depth-first search
   c. Uniform-cost search

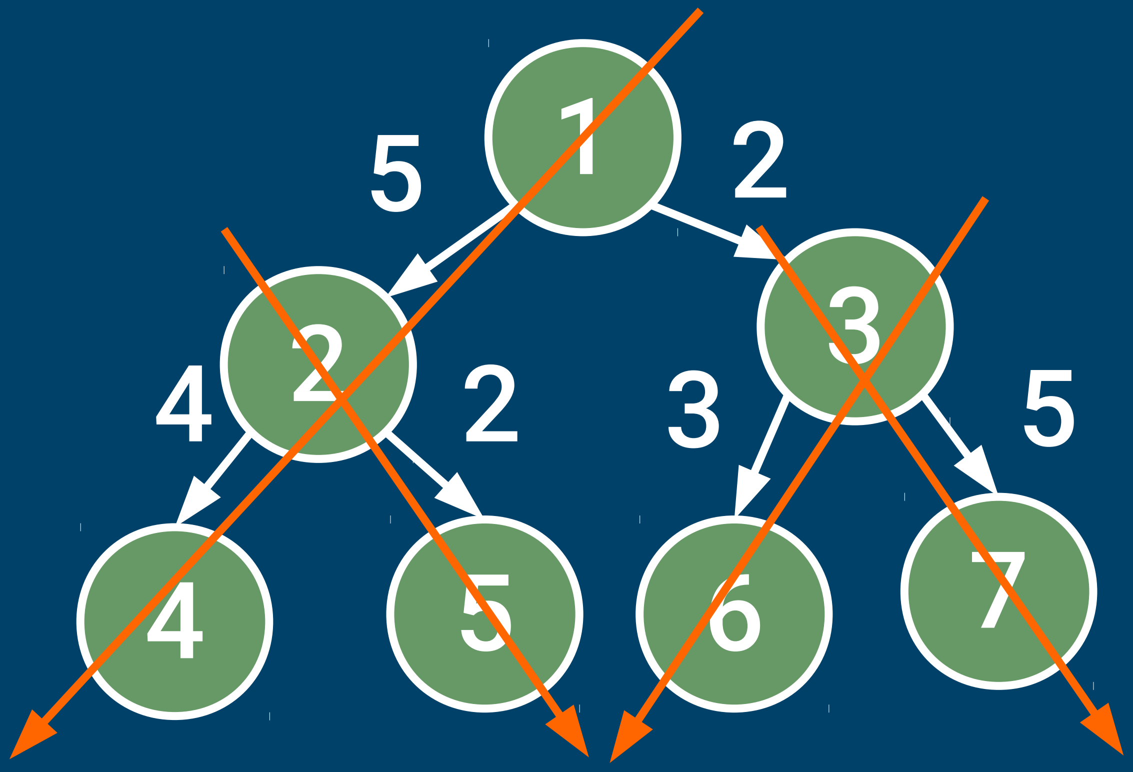
QUIZ (CONT.)

4. If the **state space** is **infinite**, which of the following is/are **complete**?
   a. Breadth-first search
   b. Depth-first search
   c. Uniform-cost search

ANSWERS

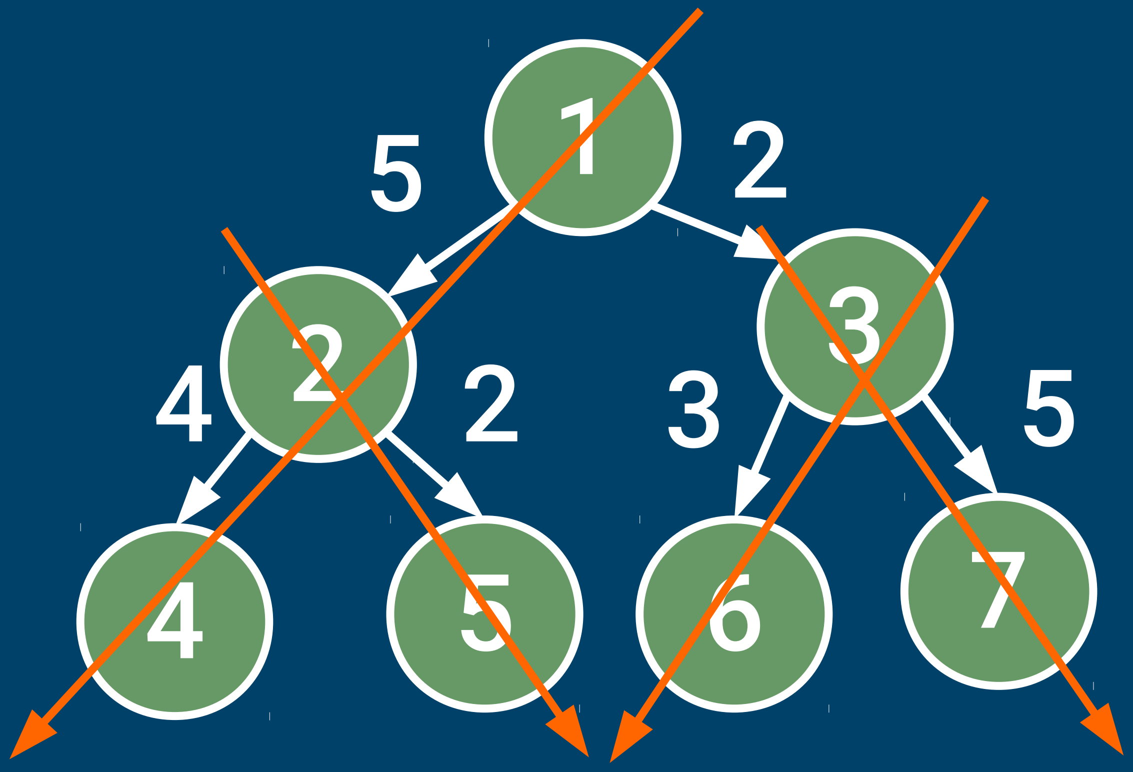
1. a. 1, 2, 3, 4, 5, 6, 7
    b. 1, 2, 4, 5, 3, 6, 7
    c. 1, 3, 6, 2, 7, 5, 4
2. Optimal for shortest path: Breadth-first search
3. Optimal for cheapest path: Uniform-cost search
4. Complete: Breadth-first & uniform-cost search

What's the problem with DFS?



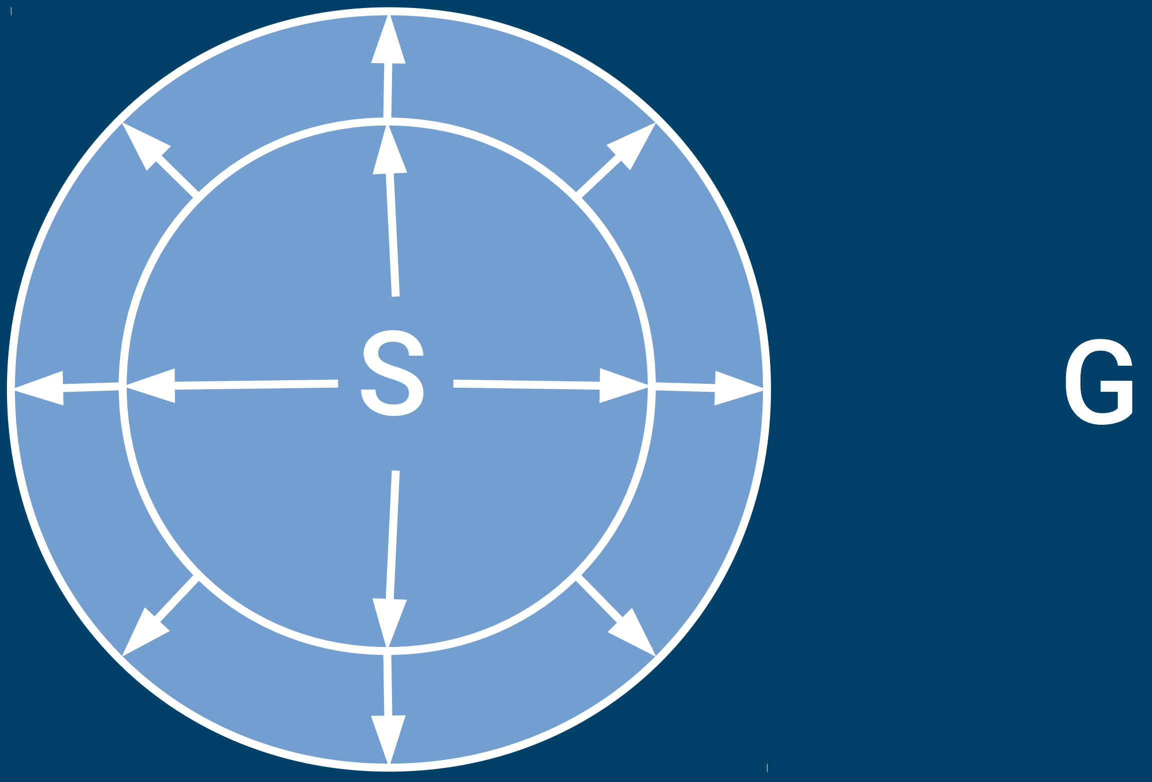
It may find a solution deeper in a subtree even if another solution can be found at a lower depth in a different subtree.

What's the problem with DFS?



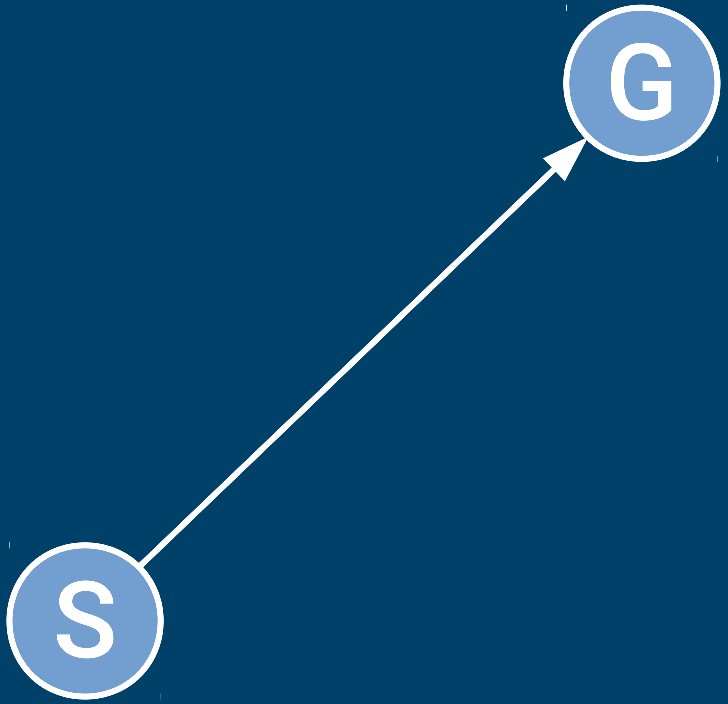
If the state space is infinite, it will get **stuck** down a subtree and never get out.

The problem with uniform-cost search is that its search is **not directed towards the goal**.



To make agents **smarter**, we need to give them **more information**; in general, the most effective info is an **estimate** of a **state's distance to the goal**.

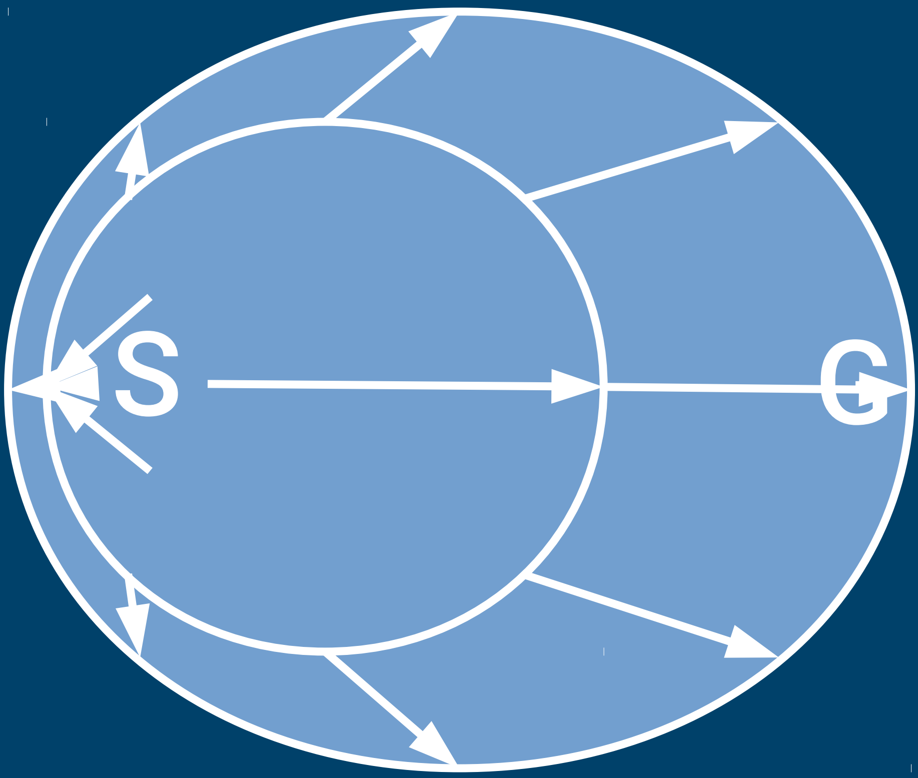
An example of a distance measure is **Euclidean distance**, i.e., **straight line distance**.



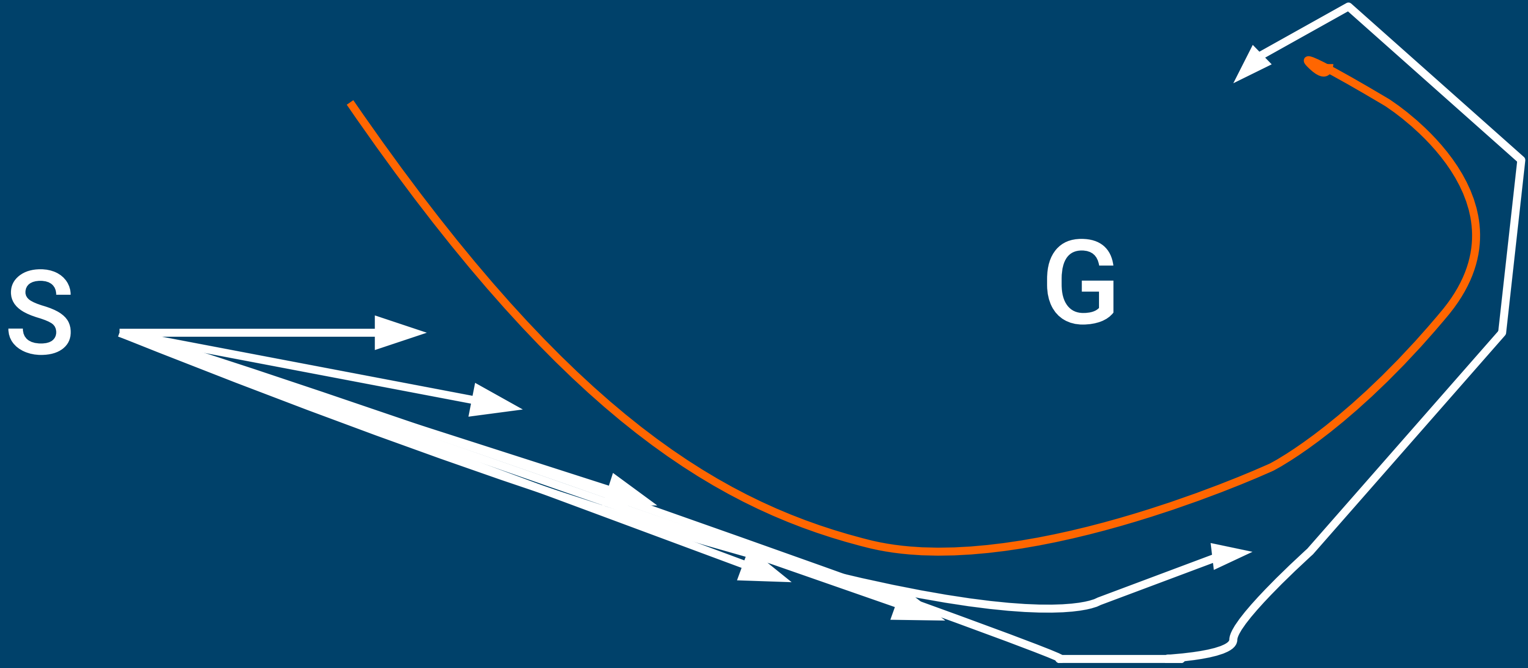
Greedy Best-First Search

Tree search algorithm that chooses the **path with closest to the goal** among unexplored paths in the frontier.

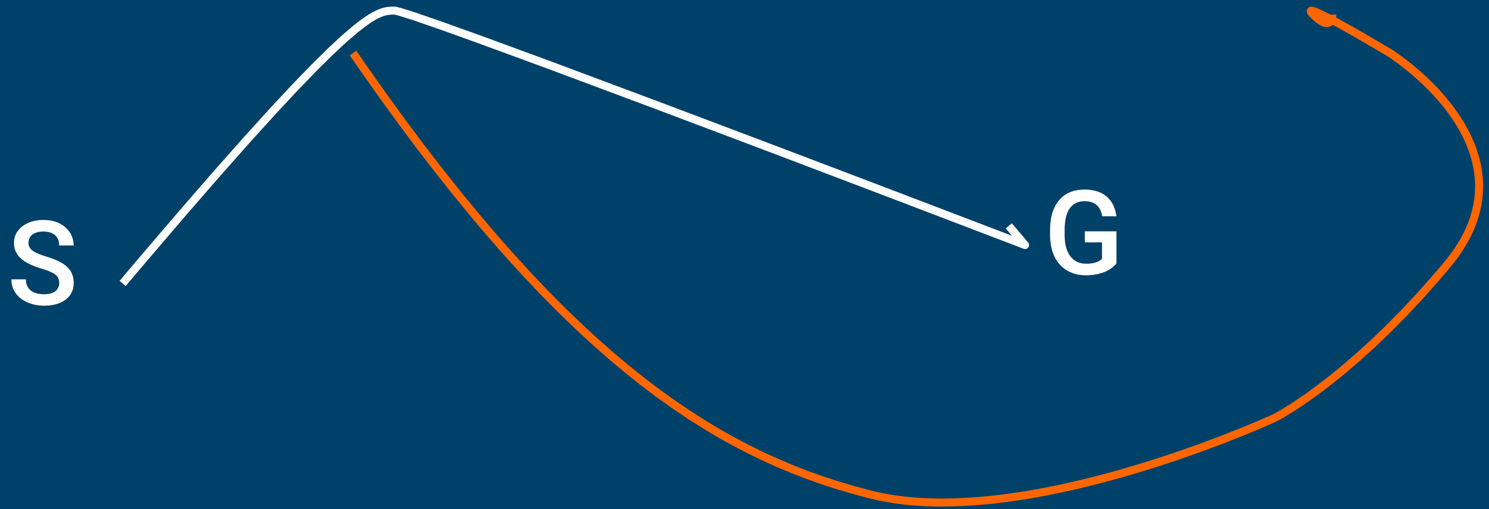
The search for the goal state thus becomes:



What if?



But this is better.



A **better search algorithm** can **find the best path** despite **expanding** only a **small number of nodes**.

A*-Search Algorithm

Tree search algorithm that chooses the **path that has a minimum value of the function F** among unexplored paths in the frontier, where

$$f = g + h$$

AND...

`g(path)` = path cost

`h(path)` = estimated remaining distance to the goal

$$S \xrightarrow{g(x)} X \xrightarrow{h(x)} G$$

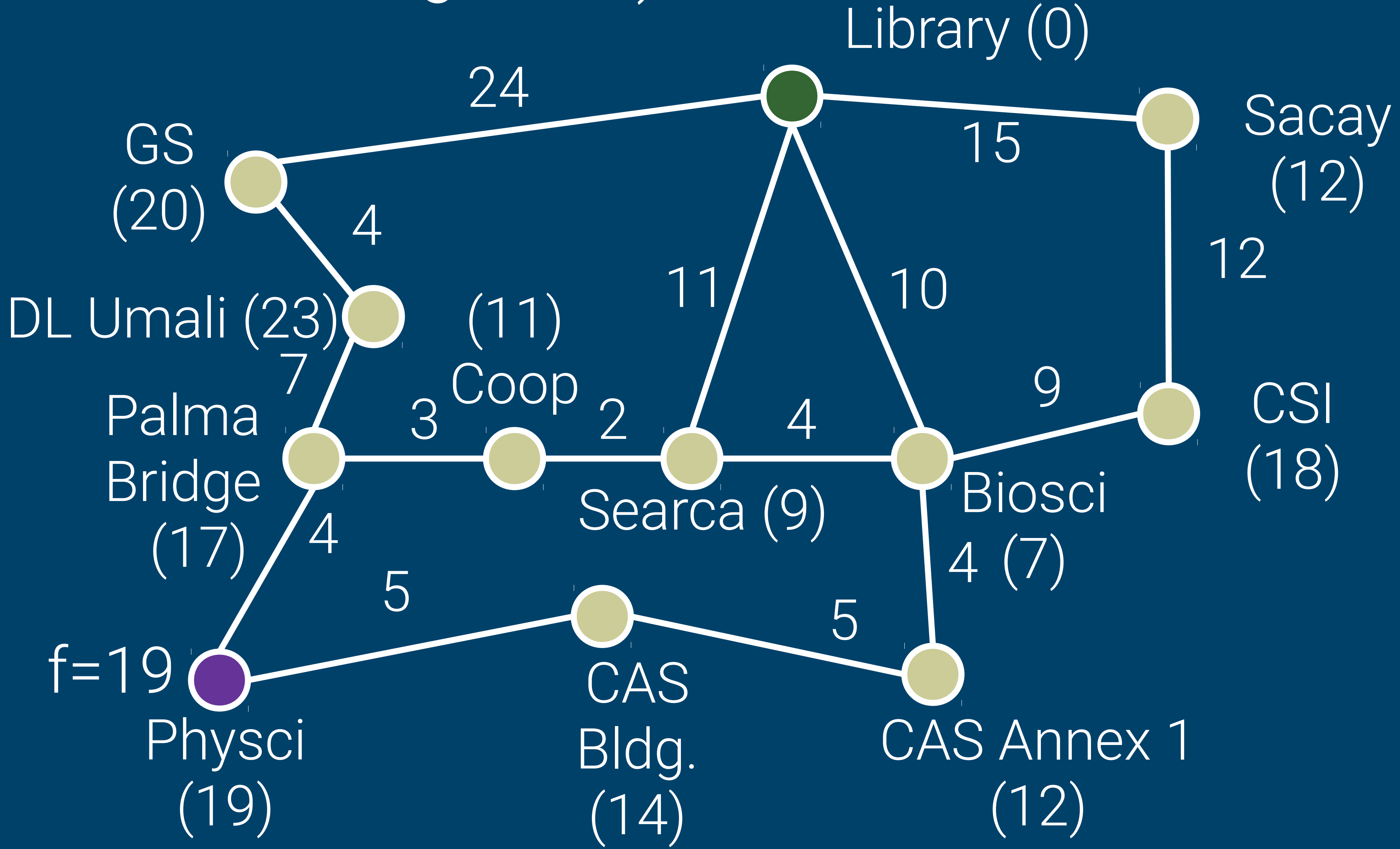
A*-search employs an open list and a closed list, which are analogous to the frontier and explored lists, respectively.

open list → frontier

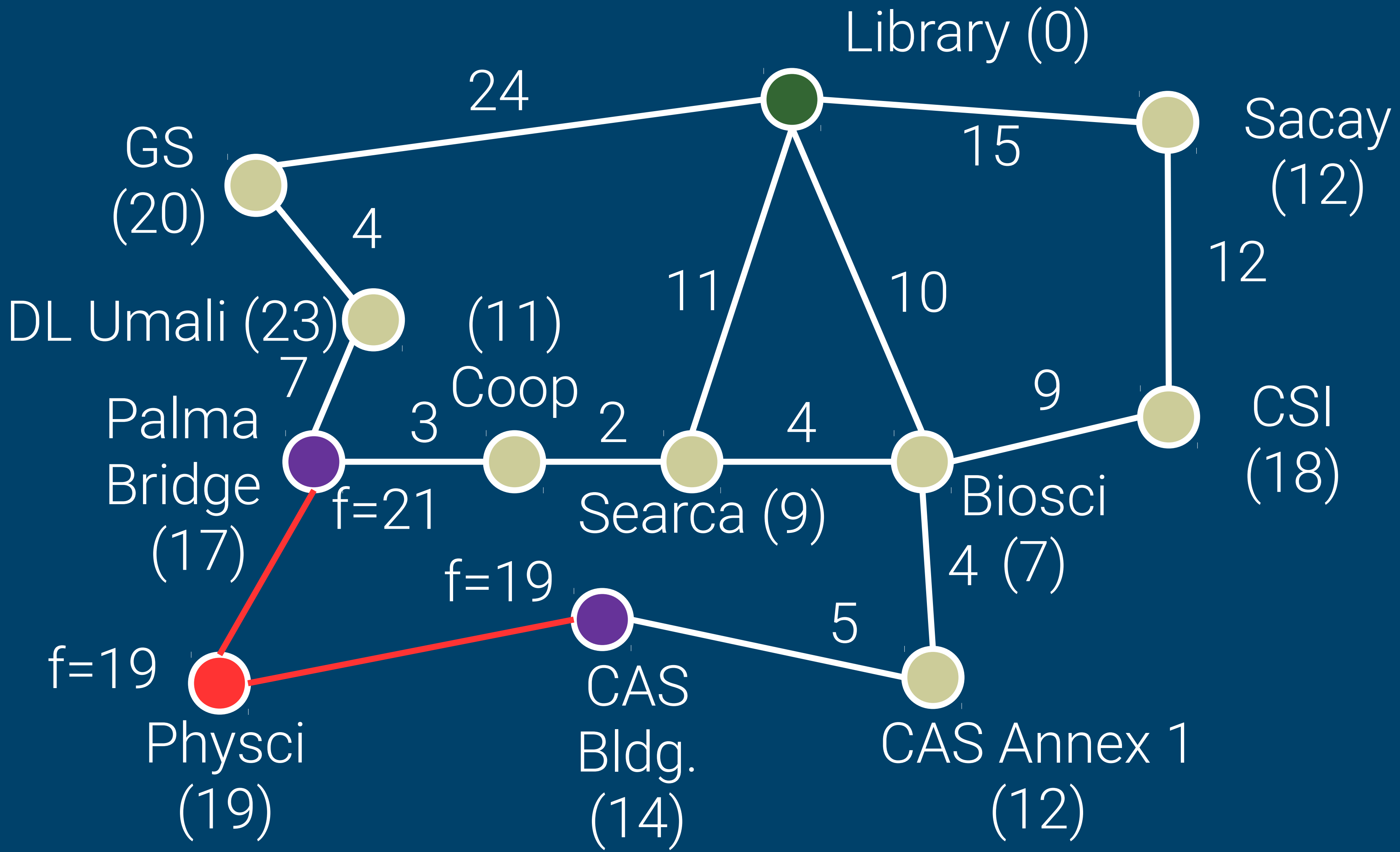
closed list → explored

```
function AStarSearch(problem) {
  openList={initial}; closedList={};
  while(openList is not empty) {
    bestNode=openList.removeMinF();
    closedList.add(bestNode);
    if(GoalTest(bestNode)) return bestNode
    for(a in Actions(bestNode)) //expand path
      if(Result(s,a) is (∉ openList or ∉ closedList)
         or ((∈ openList or ∈ closedList) and
         Result(s,a).G < duplicated.G)) {
           Result(s,a).setParent(bestNode);
           openList.add(Result(s,a));
}}}
```

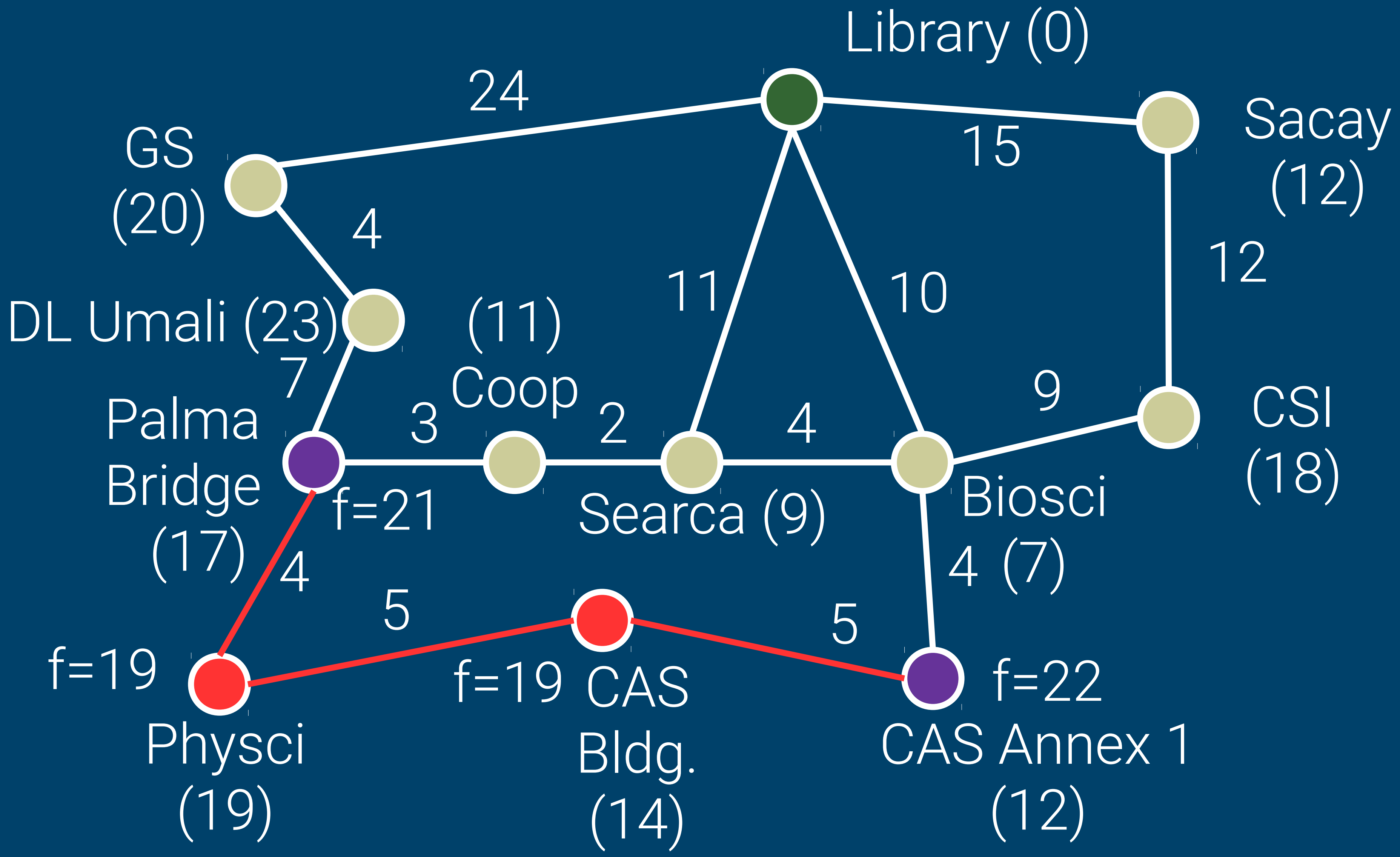
EXAMPLE. Add the initial state first (path length = 0).



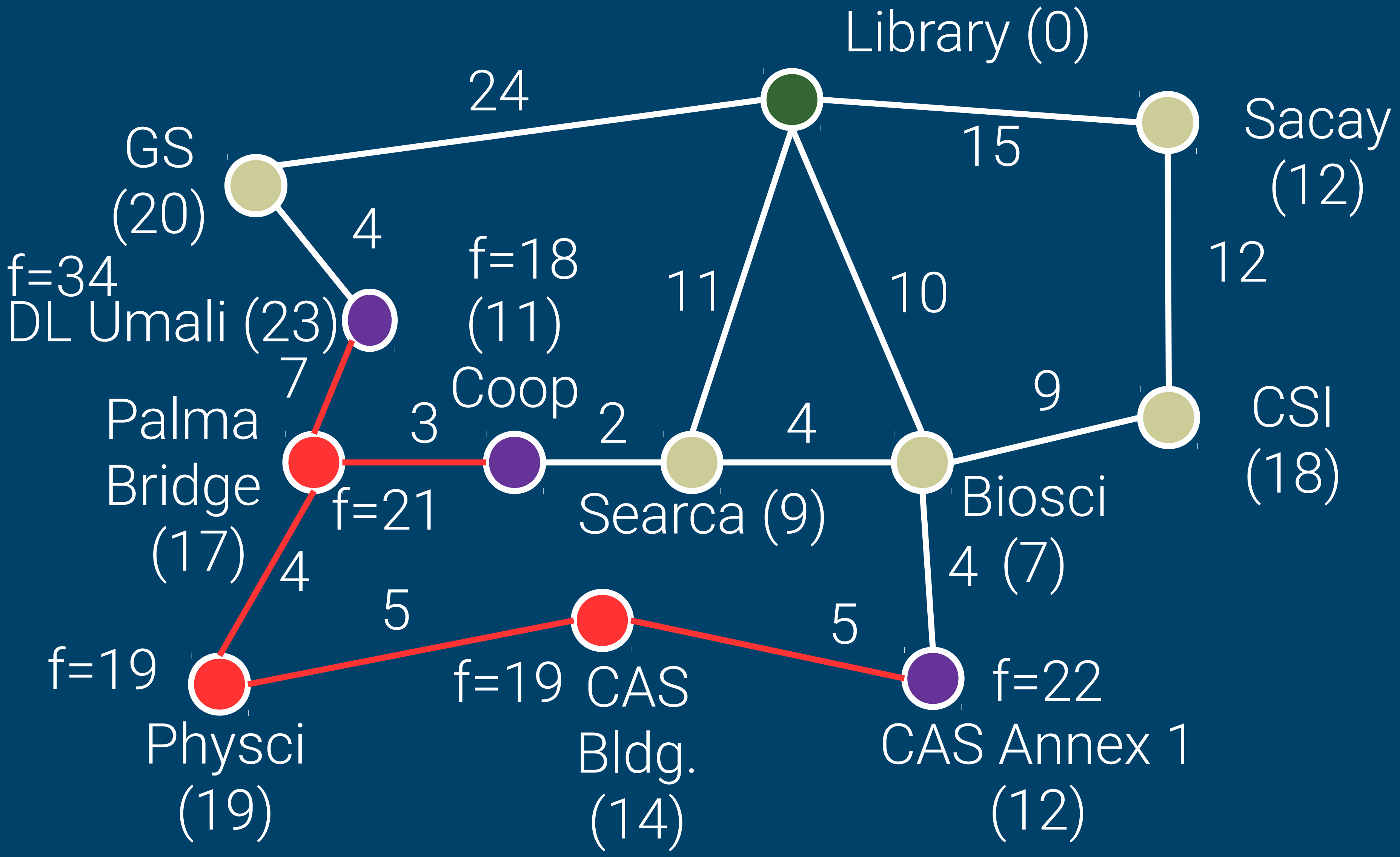
EXAMPLE. Remove Physci and add CAS Bldg. and Palma Bridge.



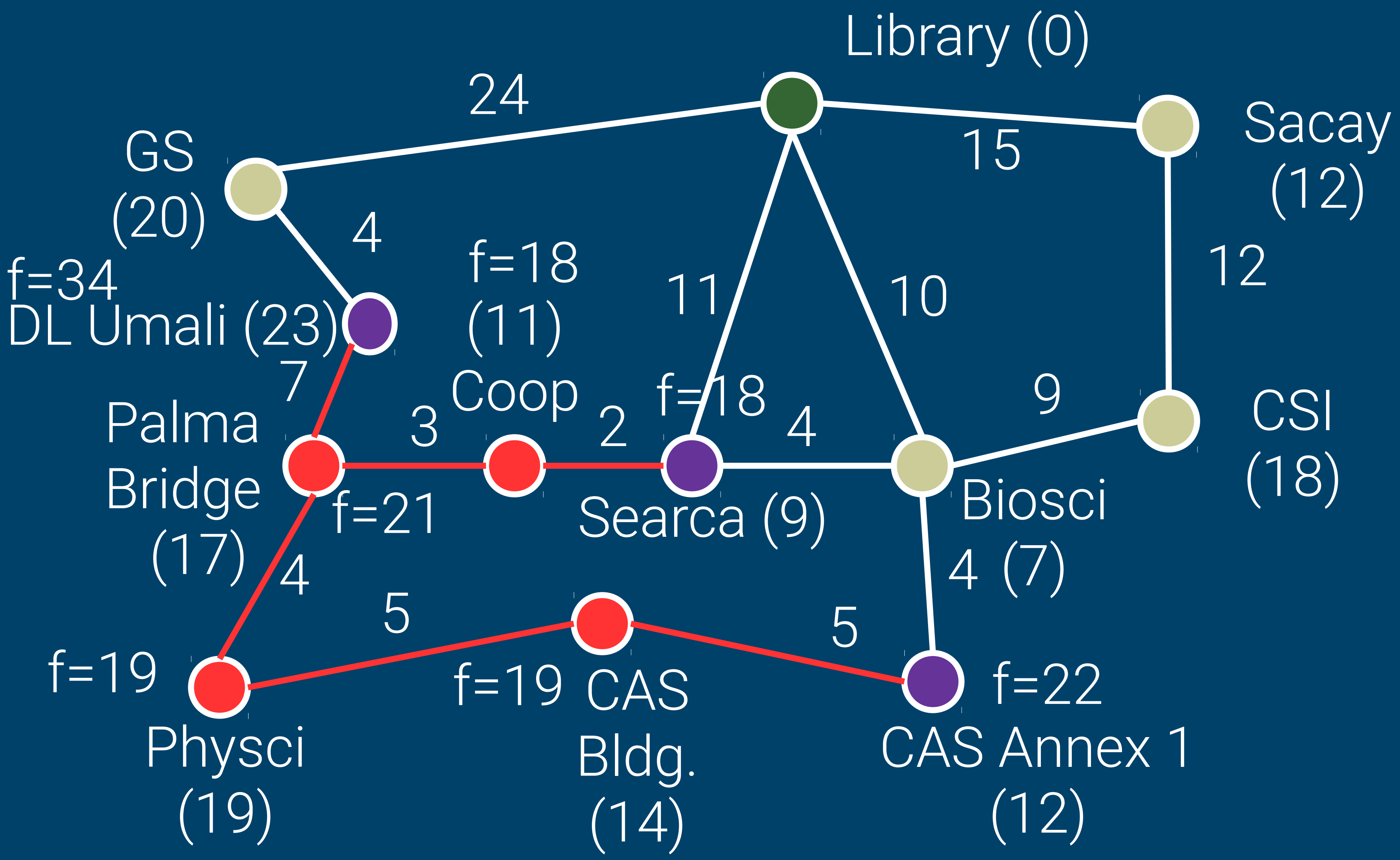
EXAMPLE. Remove CAS Bldg. and add CAS Annex 1.



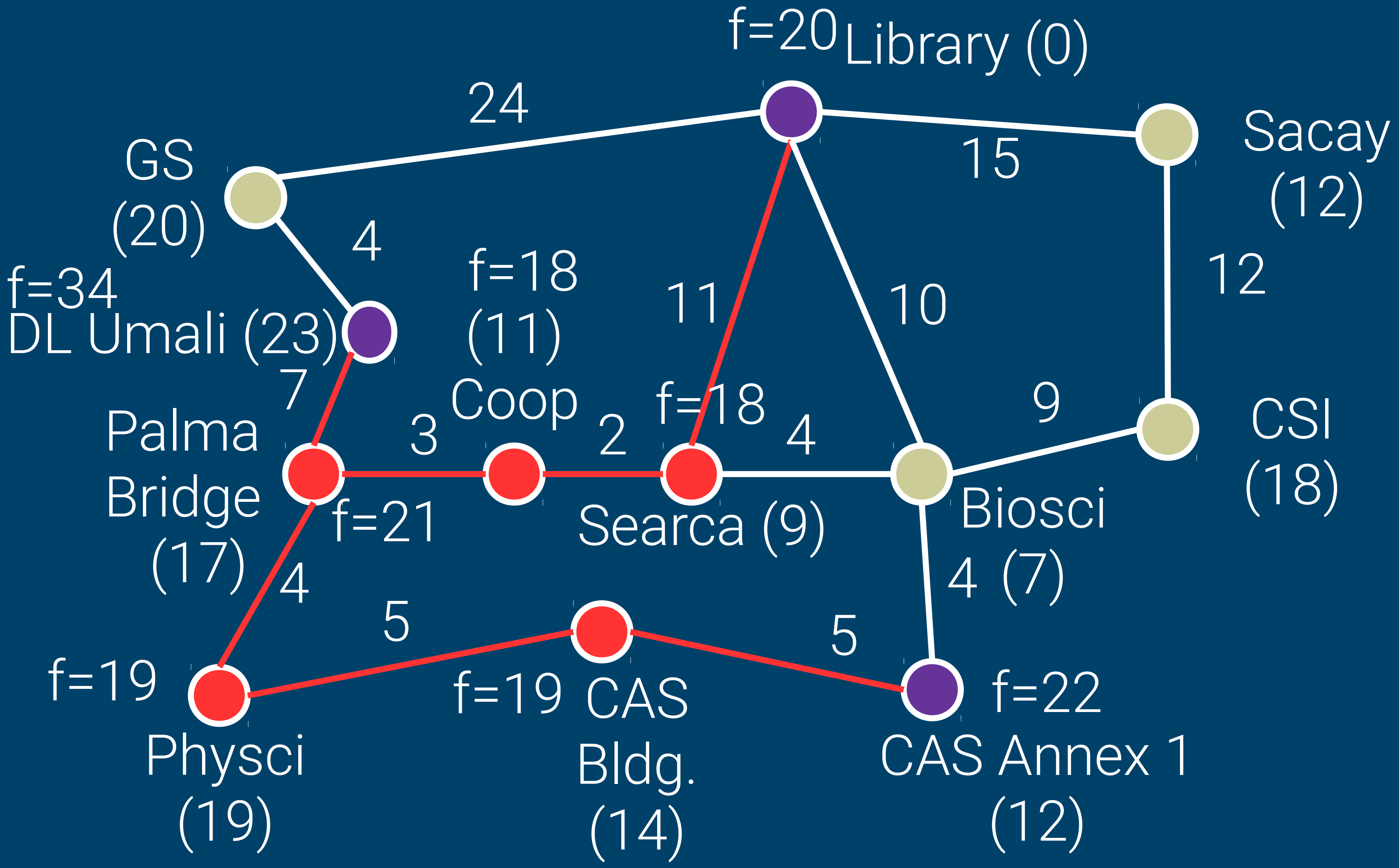
EXAMPLE. Remove Palma Bridge and add Coop and DL Umali.



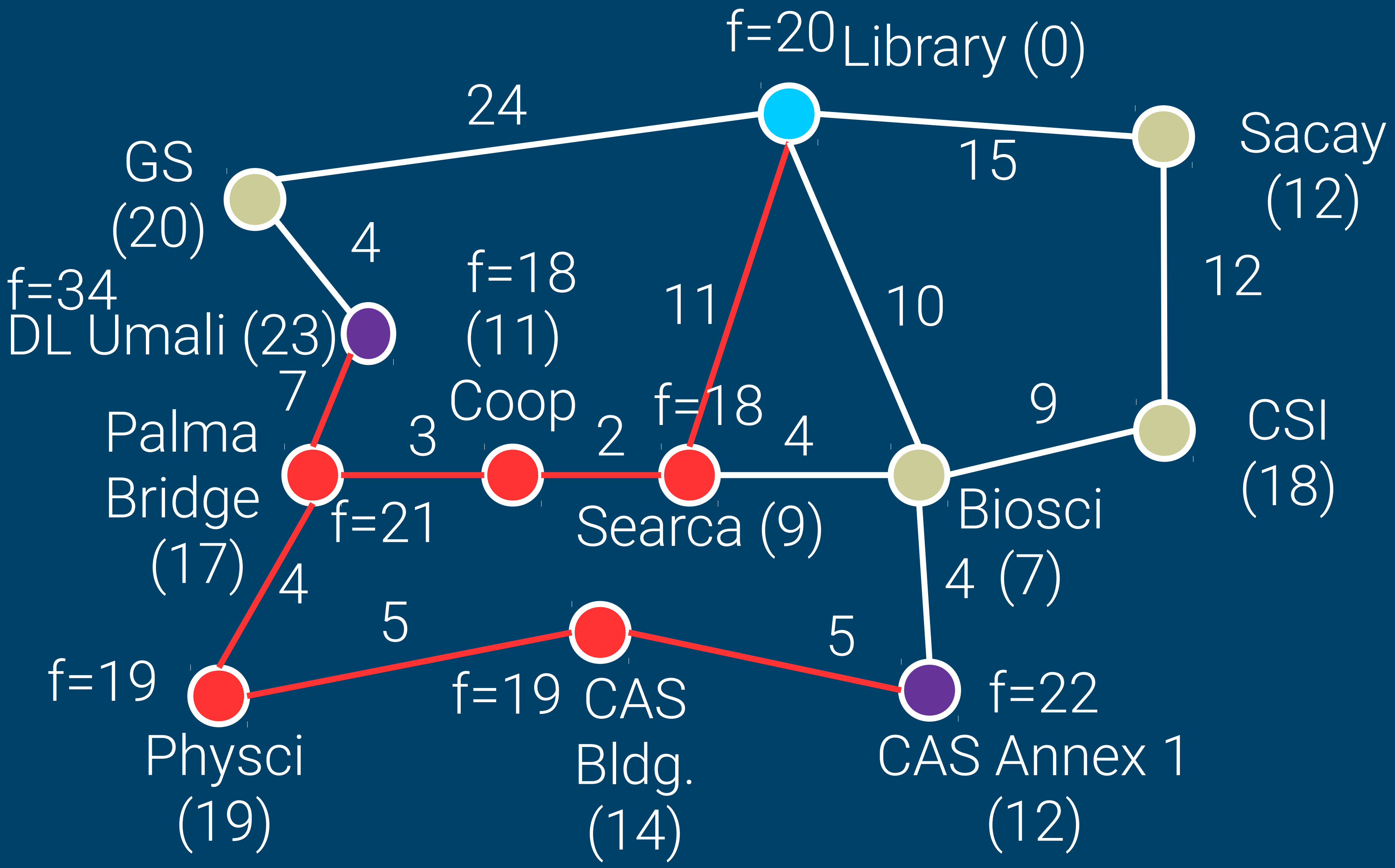
EXAMPLE. Remove Coop, add Searca.



EXAMPLE. Remove Searca, add Library.



EXAMPLE. Remove Library, end.



We have found the optimal path, with a (relatively) small number of steps.

QUIZ (CONT)

4. Will A*-search always be able to find the optimal path?
    a. Yes, always
    b. No, depends on the problem
    c. No, depends on h

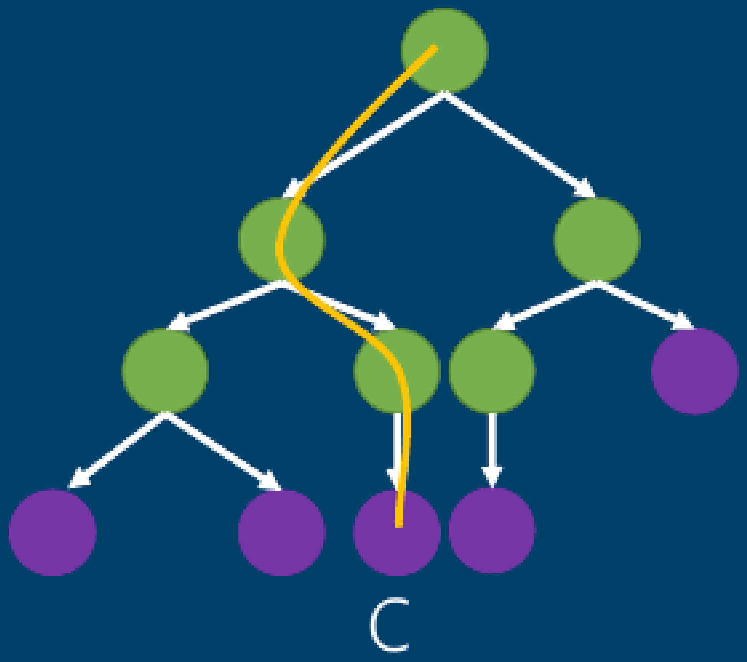
ANSWER

A*-search's ability to find the optimal solution is **based on the h function**.

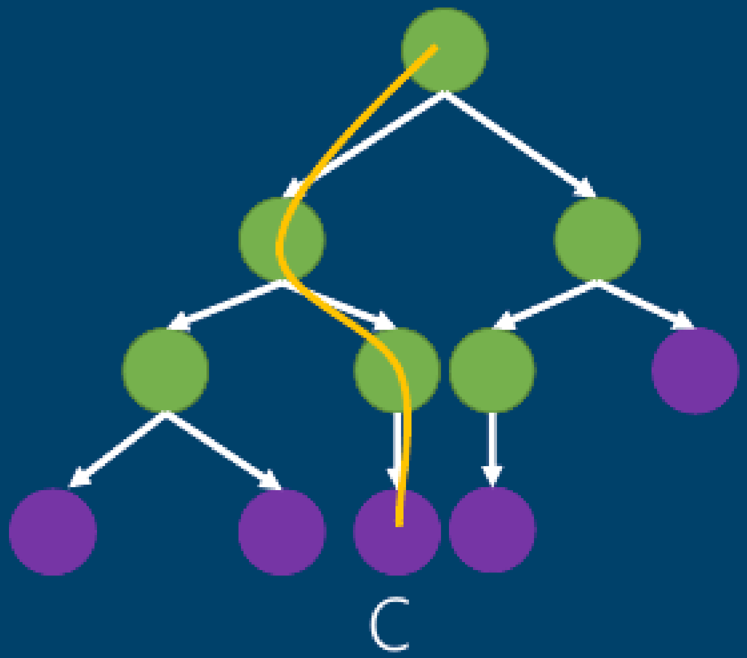
In order to find the optimal path, A*-search must use an h, such that:

$$h(path) < trueCost$$

In other words, h is **optimistic** in that it **never overestimates** the remaining distance to the goal, making it an *admissible heuristic*.



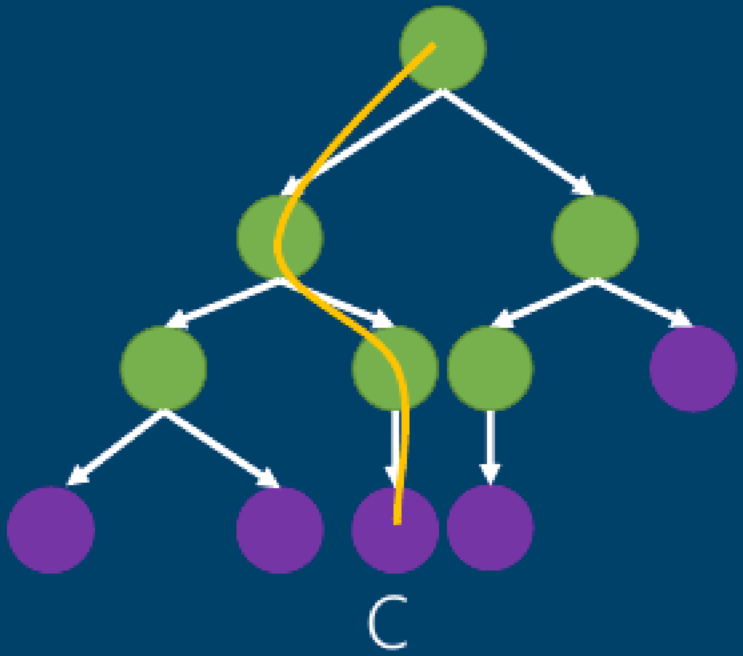
Say we have found the cheapest path with cost c.



At this point, f = c, h = 0, and g = c.

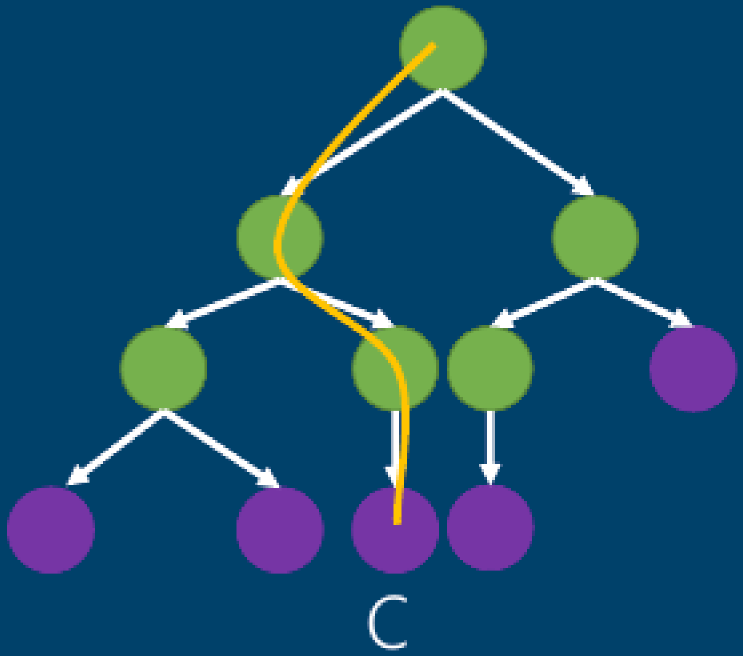
Remember,

f = g + h!

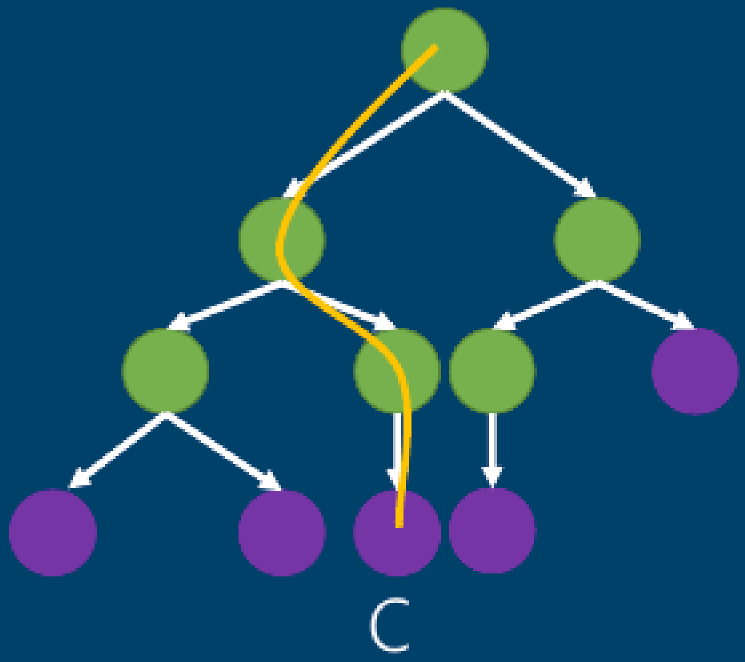


Since h is admissible,

h(path) < c.



Since A* expands the frontier with minimum f, we know that all other nodes on the frontier have f > c.



Since the heuristic is optimistic, that means that the true costs of the other nodes are also greater than c.

Tree search algorithms are not limited to solving route-finding problems.

EXAMPLE

Consider the vacuum world.

EXAMPLE

The robot can be in one of two positions, L or R.

EXAMPLE

Each position can either have dust or not.

EXAMPLE

The robot can clean the position it is in to remove the dust.

QUIZ (1/4)

1. How many states does this problem's state space have?

ANSWER

There are 8 states!

2 positions

X

2 left dust state

X

2 right dust state

ANSWER

QUIZ (¼)

3. What if we had **10 positions**, the **robot** can be **on**, **off** or **asleep**, has a **camera** that can be **on** or **off**, and has a **brush** that has **five positions**? **Each position can still have dust**, and the **robot** can **move** to **each position** and **clean it**. How many states are in the state space?

ANSWER

3 robot states x **2** camera positions x **5** brush positions x **2^{10}** positions that can be dirty or clean x **10** robot positions, giving a grand total of **307,200** states.

It is not ideal to solve such a problem using tree search algorithms, which begs the question:

When does problem solving work?

1.

The environment must be **fully-observable**.

2.

The **domain** (set of available actions) must be **known**.

3.

The **domain** must be **discrete**, that is there are a **finite number of actions** to choose from.

4.

The **domain** must be **deterministic**, that is, the **result** of taking an action can be **computed**.

5.

The **domain** must be **static**, that is, **only** the **agent** can **change** the **environment**.