UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA QUÍMICA, BIOTECNOLOGÍA Y MATERIALES

CARACTERIZACIÓN *IN SILICO* DE UNA CEPA DE *STREPTOMYCES SP.* DEL DESIERTO DE ATACAMA

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN BIOTECNOLOGÍA

DAGOBERTO ANDRÉS BOISIER SALINAS

PROFESOR GUÍA:
JUAN A. ASENJO DE LEUZE

PROFESOR CO-GUIA:
BARBARA ANDREWS FARROW

COMISIÓN:
ZIOMARA GERDTZEN HAKIM

SANTIAGO DE CHILE
2019

RESUMEN DE LA MEMORIA PARA OPTAR AL TÍTULO DE: Ingeniero Civil en Biotecnología
POR: Dagoberto Andrés Boisier Salinas
FECHA: 02/05/2019
PROFESOR GUÍA: Juan Asenjo de Leuze

CARACTERIZACIÓN *IN SILICO* DE UNA CEPA DE *STREPTOMYCES SP.* DEL DESIERTO DE ATACAMA

La creciente tendencia por parte de bacterias patógenas de volverse resistentes a los antibióticos ha llevado al campo de la ciencia a una incansable búsqueda por nuevas moléculas capaces de acabar con estos microorganismos. Una tendencia ha sido buscar compuestos naturales desarrollados por bacterias de ambientes extremos. En esa búsqueda, científicos hallaron una cepa de *Streptomyces* en el Salar de Huasco, región de Tarapacá, Chile, con propiedades antibióticas y citotóxicas.

Con el fin de poder comprender de mejor manera el metabolismo de esta cepa, se realizó un modelo metabólico *in silico* de tal cepa, denominada HST28, en base al genoma secuenciado de ésta. La primera fase fue determinar la filogenia de la cepa mediante comparación de rRNA 16S, lo cual reveló una cercanía del 97% con *Streptomyces avermitilis*. Posteriormente, usando la información disponible en la base de datos KEGG, se elaboró el modelo metabólico en base a la comparación entre HST28 y *S. avermitilis* mediante BLAST usando la extensión de Python COBRApy en Jupyter para armar metabolitos y reacciones.

Se generó una serie de funciones destinadas a facilitar el trabajo de elaboración del modelo, junto con un protocolo para encontrar errores en el modelo terminado.

El modelo final cuenta con 804 reacciones, 627 metabolitos y 594 genes, lo que constituye un modelo de tamaño promedio para este tipo de estructuras. El análisis al modelo determinó que el HST28 debería crecer mejor con sacarosa como fuente de carbono, y amonio como fuente de nitrógeno. Las tasas de crecimiento en medios mínimos fueron altas en comparación a otros *Streptomyces*.

Finalmente, queda destacar que el procedimiento para generar y curar el modelo tiene la potencialidad de ser útil en otras investigaciones que impliquen el desarrollo de un modelo metabólico.

Dedicado con mucho afecto a
mis padres, quienes siempre dieron
todo para sacarme adelante.

Agradecimientos

Llegando al fin de la etapa universitaria, siento enorme gratitud por muchas personas que me han acompañado en este proceso. En primer lugar, debo agradecer a mis padres, quienes me han apoyado toda la vida en cada uno de los procesos que he vivido. Sin ellos, no hubiera llegado a ningún lado. Son el pilar de mi vida, lo han sido siempre y espero que lo sigan siendo, al menos hasta que pueda devolverles la mano.

También agradezco a mis hermanos, eternos compañeros de juego a quienes quiero con el alma, y siempre lo haré.

No puedo dejar de mencionar a Constanza Becerra, Mariana Fernández, Pablo Lizama, y María Eugenia Parra, quienes me brindaron su amistad durante todos estos años de carrera y han dejado una marca en mi forma de ser. Los quiero, mucho, que lo sepan siempre.

También debo agradecer a David Sepúlveda, por su acogedora presencia en mi morada. Has sido un buen compañero estos últimos años y me has salvado de lo que sería una desoladora soledad. Muchas, pero muchas gracias.

Debo mencionar también a Valeria Razmilic por guiarme en este largo proceso que fue la tesis. Su buena disposición y amable sonrisa siempre me hicieron sentir acogido cuando tenía que ir a preguntarle hasta la más tonta de las dudas. Estaré siempre agradecido.

Y por último, como no olvidarme, te agradezco, César, por todo.

Tabla de contenido

Índice de tablas

Índice de figuras

1. Introducción y objetivos

En los últimos años, la industria farmacéutica y el campo de la medicina se han visto enfrentados a la cada vez mayor resistencia a los antibióticos por parte de microorganismos patógenos (Laxminarayan et al., 2013). Ante este escenario, una alternativa ampliamente usada ha sido buscar nuevos compuestos antibióticos en medios extremos. Por ejemplo, se ha encontrado malacidina en muestras de suelo desértico estadounidense (Hover et al., 2018); odilorrabdina en la bacteria *Xenorhabdus nematophila*, bacteria simbiótica de nemátodos (Pantel et al., 2018); y teixobactina en la *Eleftheria terrae*, encontrada con una nueva tecnología llamada iChip (Piddock, 2015).

Particularmente en el ámbito chileno, se han realizado búsquedas en el Desierto de Atacama, ubicado en el norte del país, y que se destaca por ser el desierto no polar más árido del planeta (Thomas, 1997). Se han hallado cianobacterias en la zona de Yungay (24°06’ S, 70°01’ W), que corresponde al área más seca del desierto (Lacap, Warren-Rhodes, McKay, & Pointing, 2011). También se han hallado arqueas productoras de carotenoides en la Laguna Tebinquiche (23°08’ S, 68°15’ W), caracterizada por su alta salinidad; cepas de *Streptomyces* productoras de aminobenzoquinonas (moléculas con capacidades antiinflamatorias (Schulz et al., 2011)) en el Salar de Tara (23°04’ S, 67°15’ W); entre otras actividades investigativas del área microbiológica (Bull & Asenjo, 2013).

Para analizar los microorganismos encontrados en estos ambientes extremos, se ha preferido en los últimos años el usar métodos computacionales en vez de utilizar mecanismos tradicionales (como el cultivo en placas), dado que implica un tiempo menor, y requiere un conocimiento menos acabado de la fisiología del microorganismo. Cabe mencionar que acorde con estudios, el 99% de los microorganismos existentes no son cultivables por medios tradicionales (Stach & Bull, 2005). Además, estos métodos digitales permiten procesar cantidades considerables de información en tiempos menores. Se han diseñado diversas herramientas y programas con el fin de volver más fácil, rápida y eficiente la extracción y uso de información biológica a partir de estas formas de vida.

Entre las herramientas anteriormente mencionadas, destacan los modelos metabólicos como una aproximación virtual de los mecanismos y procesos que ocurren a nivel intracelular. A grandes rasgos, el modelo metabólico es un arreglo de tres tipos de conjuntos: Reacciones, metabolitos y genes. Estas clasificaciones están interrelacionadas, de modo que cada reacción transforma metabolitos de una clase a otra, según los genes que regulan cada proceso. La función de estos modelos es predecir, o estimar, el comportamiento del sistema vivo en base a los conocimientos que se tienen del organismo. A esto se le llama caracterización *in silico* del organismo. Actualmente, existen varias aplicaciones que permiten la elaboración de estos modelos, como extensiones de MATLAB (llamada COBRA) (Becker et

al., 2007), funciones de Python (COBRApy) (Ebrahim, Lerman, Palsson, & Hyduke, 2013), y otros, como Escher, que además permiten una realizar una representación gráfica del modelo (King et al., 2015).

1.1 Objetivos generales

El objetivo principal es elaborar un modelo metabólico de la cepa HST28, un *Streptomyces* encontrado en el Salar de Huasco, en el norte de Chile, con propiedades de generar moléculas con propiedades antibióticas y anticancerosas.

1.2 Objetivos específicos

Para llevar a cabo el objetivo principal, se debe determinar un organismo de referencia cuyo genoma ya esté estudiado *in silico*, para usarlo como base a la hora de definir los genes y reacciones presentes en el modelo. Además, se debe generar una metodología de elaboración del modelo metabólico, que incluya adición de conversiones de metabolitos, adición de reacciones globales de biomasa, y adición de reacciones de transporte e intercambio con el medio externo; que funcione de manera automatizada a partir de las bases de datos disponibles. También se debe realizar una curación manual del modelo, junto con un protocolo de detección de errores de funcionamiento del mismo. Teniendo todo lo anterior, se debe determinar teóricamente las condiciones ideales para el crecimiento de la cepa en cuanto a nutrientes, particularmente fuentes de carbono y nitrógeno para generar biomasa.

2. Antecedentes generales

2.1 Salar del Huasco

El Salar del Huasco (20°18’ S, 68°50’ W) es una cuenca endorreica del desierto de Atacama, ubicada en la Región de Tarapacá, a 176 kilómetros al este de la ciudad de Iquique, a 3.800 msnm, y es parte de un santuario de la naturaleza homónimo desde 2005. El salar abarca una región de 51 km^2, de los cuales 2,5 km^2 corresponden a cuerpos de agua (Risacher, 1999).

El medio del Salar del Huasco es atalasohalino, vale decir, con una alta salinidad, pero con proporciones de sales distintas a aquellas encontradas regularmente en ambientes marinos. A pesar de la alta concentración de sales, alberga especies vivas tanto macroscópicas, como el flamenco andino (*Phoenicoparrus andinus*); como microscópicas, desde especies planktonicas hasta procariontes (Sielfield, Amado, Herreros, & Peredo, 1996). Se han encontrado arqueas (Dorador, Vila, Remonsellez, Imhoff, & Witzel, 2010), cianobacterias (Dorador, Vila, Imhoff, & Witzel, 2008), y actinomicetes, teniendo estos últimos la capacidad de secretar compuestos con propiedades antibióticas, antifúngicas y citotóxicas de interés (Bull & Asenjo, 2013).

2.2 Modelos de escala genómica y análisis de balance de flujos

Los modelos de escala genómica son representaciones computacionales de la fisiología de un organismo, escritos sobre la base de las relaciones existentes entre las moléculas, reacciones y genes. Particularmente, la idea es definir qué moléculas se convierten mediante cuáles reacciones, y qué genes regulan tal proceso, de una manera comprensible, ya sea para seres humanos o para elementos digitales.

Las principales utilidades de estos modelos son: Predecir comportamientos celulares ante diversas situaciones, estimar el efecto de modificaciones genéticas, simular interacciones intercelulares, descubrir nuevas vías productoras de metabolitos, entre otras (Oberhardt, Palsson, & Papin, 2009).

Por lo general, el primer paso para desarrollar un modelo es contar con anotaciones del genoma del organismo. Posteriormente viene conseguir información bibliográfica respecto al metabolismo del mismo. Actualmente, existen diversas bases de datos que permiten acceder expeditamente a información referente a ambos temas, tales como KEGG o BioCyc. Con los conocimientos requeridos recopilados, se puede utilizar algún programa que

facilite la incorporación de la información y la generación del modelo. Un ejemplo de programa es COBRA, que será descrito más adelante.

Una manera de procesar la información contenida en los modelos de escala genómica es mediante análisis de balance de flujos (FBA, por sus siglas en inglés), el cual es, a grandes rasgos, una optimización del modelo visto como un problema lineal. Particularmente, un modelo puede ser interpretado como una matriz de *m* por *n* elementos, donde *m* es en números de metabolitos y *n* el número de reacciones. La matriz se rellena con los coeficientes estequiométricos de cada metabolito para cada reacción, colocando ceros en los casos en donde no haya participación de un determinado metabolito en tal reacción. Luego, se completa el problema mediante la definición de un objetivo, que generalmente es la maximización de un flujo a través de una reacción. Este problema de optimización lineal puede ser resuelto mediante softwares o funciones especializadas, entegando una lista de flujos de cada reacción, lo cual puede ser analizado para entender de mejor manera el comportamiento del sistema a modelar.

2.3 KEGG

La Kyoto Encyclopedia of Genes and Genomes, abreviada KEGG, es una base de datos desarrollada por la Universidad de Kyoto, Japón, orientada a la comprensión de los sistemas biológicos en base a su información a nivel genómico y molecular. Para ello, consta de un amplio bagaje de datos referentes a genes y genomas de especies tanto eucariontes como procariontes, y relaciones de éstos con metabolitos intra y extracelulares. Toda esta información está entrelazada en redes metabólicas, que permiten entregar de manera comprensible el conocimiento que está detrás de cada una de ellas. Un ejemplo se ve en la Figura 1, donde los puntos corresponden a los metabolitos, las flechas son las reacciones, y los cuadros son enlaces que conectan información más detallada, ya sea sobre la reacción, los genes involucrados, o la enzima utilizada.

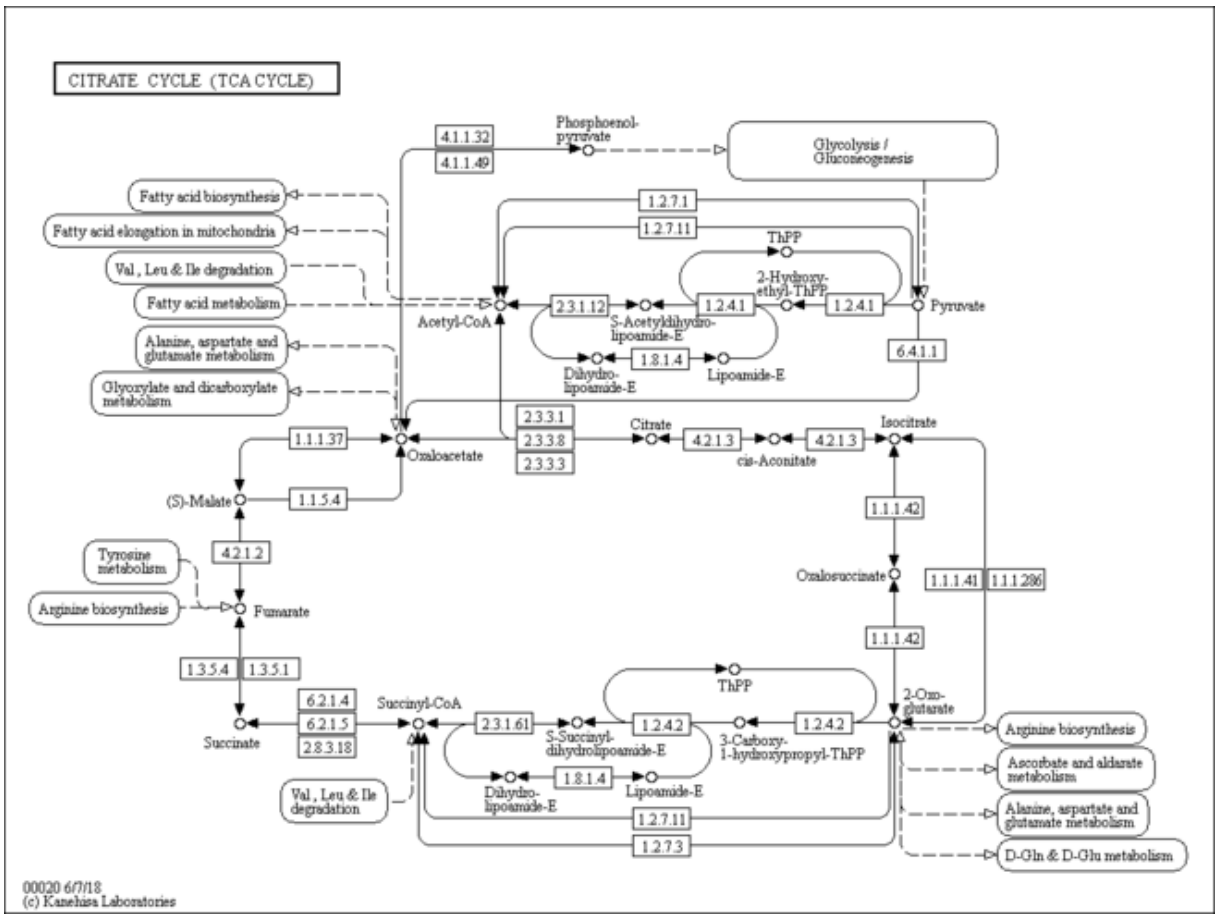

Figura 1. Ejemplo de mapa metabólico de KEGG.

Presentado por primera vez en 1995, KEGG ha ido aumentando gradualmente los servicios de información que dispone al público. En un principio, KEGG presentaba mapas, metabolitos, genes y enzimas. En 1998 se agregó la base de datos de reacciones. A partir de 2000, se empezó a incluir genomas completos, accesibles mediante un código para cada organismo. En 2002, se agregó el concepto de ortologías: Grupos de genes de diversas especies que codifican para la misma reacción. Así, KEGG ha agregado más variedad a su base de datos con el paso del tiempo, siendo las últimas adiciones en el campo de la salud humana, como la suma en 2017 de variantes de genes de *Homo sapiens* (Kanehisa, 2019b).

Actualmente, KEGG tiene en su haber más de 18.000 metabolitos, más de 11.000 reacciones, y alrededor de 29.000.000 de genes agrupados en más de 6.000 genomas (Kanehisa, 2019a), siendo una de las bases de datos más importantes del área de las ciencias de la vida.

2.4 Bioservices

Bioservices, desarrollada en 2013, es una extensión para Python que permite acceder a diversas bases de datos de información ligada al ámbito de la biología, tales como KEGG, ChemSpider y BioDBnet, entre otras (Cokelaer et al., 2013). Actualmente, Bioservices contiene aplicaciones para

acceder a 33 bases de datos públicas de internet del área de las ciencias biológicas.

El desarrollo de investigación en el área de la biología y biotecnología está caracterizado por requerir el uso de varias fuentes de información, y generalmente es necesario tomar datos varias veces a partir de la misma fuente, lo cual puede tomar una excesiva cantidad de tiempo, ralentizando el desarrollo. Bioservices es una respuesta ante esta problemática, y permite acceder de manera expedita a grandes volúmenes de información disponible en internet, junto con entregar herramientas para procesar y facilitar la comprensión de aquellos datos.

Por ejemplo, para el caso de la interacción con la base de datos KEGG, Bioservices ofrece la clase `KEGG`, dotada de funciones tales como `find` para encontrar componentes de la base de datos que cumplan con los requerimientos entregados por el usuario; la función `get`, que permite obtener y descargar información de la base de datos usando el o los identificadores proporcionados por KEGG; y `list`, que entrega una lista de todos los identificadores para cierta variable asociada a un organismo, entre otras.

Adicionalmente para KEGG, Bioservices ofrece la clase `KEGGParser`, la cual, mediante la función `parse`, permite subdividir la información entregada por `get` a un formato manejable por el usuario final.

Bioservices realiza operaciones similares a las descritas para las bases de datos que procesa, entregando un potente recurso a la hora de buscar información en la red en temas de biología y biotecnología.

2.5 COBRA

El método de análisis y reconstrucción basado en restricciones (en inglés, COnstraint-Based Reconstruction and Analysis) fue diseñado en 2007, respondiendo al avance del desarrollo de la biología de sistemas, particularmente en lo que es construir y validar modelos *in silico* de fenómenos biológicos (Becker et al., 2007). Su funcionamiento principal se basa en la conceptualización de especies (Ej: moléculas) que interactúan dentro o en el entorno inmediato de la célula a estudiar. Tales interacciones, que pueden ser reacciones o procesos de transporte, se describen en matrices estequiométricas. Éstas son posteriormente asociadas a matrices que marcan los límites superiores e inferiores de las velocidades de cada interacción, que vendrían siendo las restricciones del modelo. Posteriormente, a través de algoritmos de optimización, es posible resolver el problema propuesto, cuya representación biológica es el crecimiento teórico del organismo analizado, o la producción de algún metabolito de interés por parte de éste.

En 2011, se desarrolló una versión mejorada de COBRA, que incluye, entre otras nuevas capacidades, estimaciones de flujos metabólicos y opciones

de visualización de redes metabólicas (Schellenberger et al., 2011). El método COBRA está originalmente diseñado para operar con el programa de análisis matemático MATLAB; sin embargo, en 2013 se diseñó una versión de COBRA compatible con el lenguaje de programación Python, el cual fue denominado COBRApy (Ebrahim et al., 2013). Este último contiene todas las funcionalidades de COBRA mejorada, con la excepción de la expresión gráfica de resultados. Cabe mencionar que COBRApy está sustentado sobre una plataforma gratuita (Python), a diferencia de MATLAB, que es pagado. Este hecho permite abrir el acceso a todo el público con intereses en el área, además de alivianar el costo de la investigación en biología y biotecnología.

2.6 BLAST

El programa BLAST (Basic Local Alignment Search Tool) es una herramienta desarrollada por el Centro Nacional para la Información Biotecnológica de Estados Unidos (NCBI) en 1989, la cual permite encontrar similitudes entre secuencias genómicas. Básicamente, el programa compara una secuencia entregada por el usuario con una base de datos, entregando como resultado una serie de secuencias de ésta que tienen regiones en común con aquella ingresada ("BLAST® Command Line Applications User Manual," n.d.). Usualmente, BLAST se lleva a cabo en línea, debiendo el usuario subir su secuencia problema a la página del NCBI (https://blast.ncbi.nlm.nih.gov) para que el programa compare contra alguna de las bases de datos propias del Centro.

La importancia de esta herramienta radica en la naturaleza modular de las proteínas y los genes que la codifican. Vale decir, regiones de proteínas que cumplan la misma función tendrán una secuencia aminoacídica parecida. Por ende, es posible determinar el rol que tiene un gen en un sistema en base a su similitud con alguna otra secuencia genómica conocida. En este sentido, BLAST juega un papel crucial en la elaboración de modelos metabólicos, en cuanto permite tener nociones de la funcionalidad de los genes de un ser vivo desconocido en base al conocimiento del genoma de otros organismos parecidos.

A partir de esta aplicación, se han creado mejoras que responden a satisfacer necesidades más amplias. Particularmente, está BLAST+, que es una versión descargable de este programa, el cual permite llevar a cabo búsquedas de secuencias de manera local, sin necesidad de subir información a internet y con la posibilidad de usar bases de datos que estén almacenadas en el computador del usuario. Otra modificación es BLASTp, que usa secuencias aminoacídicas en vez de genómicas para llevar a cabo la tarea, y BLASTx, que permite realizar comparaciones entre secuencias proteicas y de nucleotídicas, mediante un traductor que interconvierte ambos tipos de información.

2.7 Los *Streptomyces*

Los *Streptomyces* corresponden a un género de bacterias englobado en el filo de las actinobacterias. Entre sus características comunes está el ser gram-positivas y tener un alto contenido de GC en sus genomas (Madigan, 2005). Están distribuidos a nivel mundial, encontrándose en diversas condiciones, tanto mesófilas, como extremas de temperatura, humedad, salinidad y acidez (Zenova, Manucharova, & Zvyagintsev, 2014).

En la naturaleza, se suelen encontrar en los suelos, siendo los responsables en gran medida del olor a "tierra húmeda" gracias a la liberación de geosmina (Zaitlin & Watson, 2006). Además de este compuesto, los *Streptomyces* son conocidos por generar una amplia variedad de metabolitos secundarios, habiendo sido registrados hasta la fecha alrededor de 10.000 compuestos producidos por estas bacterias (Zengler, Paradkar, & Keller, 2004).

Entre las moléculas generadas por los *Streptomyces*, destacan diversos antibióticos, como lo son la neomicina del *S. fradiae* (Waksman & Lechevalier, 1949), el cloranfenicol del *S.venezuelae* (Ehrlich, Bartz, Smith, Joslyn, & Burkholder, 1947), la estreptomicina del *S. griseus* (Schatz, Bugie, & Waksman, 1944), la kanamicina del *S. kanamyceticus* (UMEZAWA et al., 1957), entre otros.

Adicionalmente a la producción de antibióticos, los *Streptomyces* son capaces de generar moléculas antitumorales, como lo son las antraciclinas, moléculas que se intercalan en el DNA de la célula tumoral, impidiendo su replicación (Bakker, van der Graaf, Groen, Smit, & De Vries, 1995). Un ejemplo de antraciclinas son la daunorubicina y la doxorubicina, sintetizadas por *S. peucetius* (Lomovskaya et al., 1999).

Entre las diversas bacterias que conforman este grupo, destaca el *S. coelicolor* por tener su genoma secuenciado desde 2002 (Bentley et al., 2002), lo cual ha permitido un estudio profundo de las propiedades metabolizadoras de los *Streptomyces* en general usando a esta bacteria en particular como modelo.

3. Metodología

3.1 Determinación de organismo base

Con el fin de determinar si una reacción está presente en el metabolismo del HST28, era necesario comparar sus genes con algún organismo cuyo genoma y funciones de éste fueran conocidas. Para ello, se estudió la genealogía del HST28 usando su secuencia de RNA ribosomal 16s utilizando el servicio web EZBioCloud (Yoon et al., 2019). Los resultados más parecidos eran *Streptomyces kanamiceticus* y *Streptomyces aureus* entre otros, como se ve en la Figura 2. Sin embargo, los organismos que aparecen en el árbol filogenético no habían sido secuenciados, por lo que se optó por buscar entre los resultados de identidades más bajas a alguna especie que sí estuviera secuenciada.

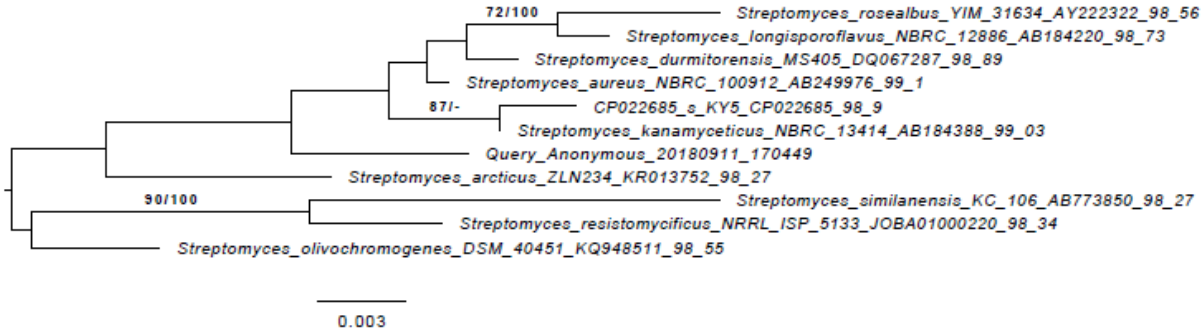

Figura 2. Árbol filogenético de las especies más cercanas a HST28. Esta última aparece como Query_Anonymous

El Anexo 8.1 muestra todos los resultados obtenidos en el estudio filogenético, destacando con amarillo el organismo secuenciado más cercano al HST28, que resultó ser el *Streptomyces avermitilis*, con un 97,9% de coincidencia. El *Strepotmyces fulvissimus* tiene el mismo porcentaje de similaridad que *S. avermitilis* y también está secuenciado, sin embargo, no se optó por este por menor disponibilidad de información. En la Figura 3 se aprecia la filogenia del HST28 en relación a estas dos especies.

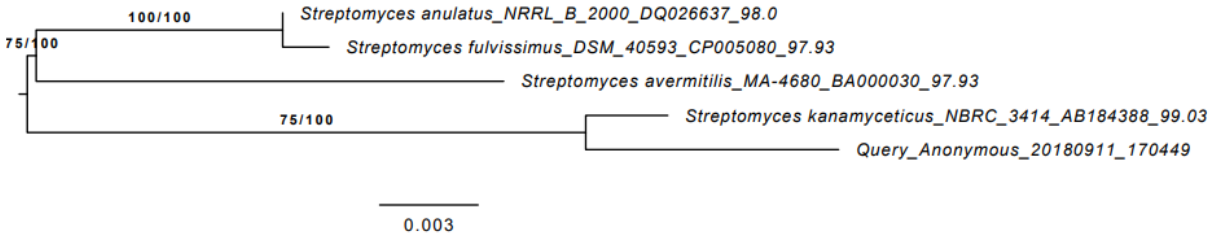

Figura 3. Árbol filogenético del HST28 con especies seleccionadas. Ésta aparece como Query_Anonymous

3.2 Elaboración de modelo metabólico

3.2.1 Adición de reacciones de conversión metabólica

Para construir el modelo, se analizaron las vías metabólicas conocidas de *Streptomyces avermitilis* disponibles en KEGG, para luego seleccionar los genes que regulan las reacciones de interés para el modelo. Tales genes fueron descargados de la base de datos de KEGG, para luego ser comparados con el genoma del HST28 mediante BLAST de proteínas local. Si la comparación arrojaba como resultado la presencia de un gen con función homóloga en el sistema, se agregaba la reacción asociada al modelo. Este proceso fue automatizado mediante un código en Python que, al ingresarle los números de serie de los genes de *S. avermitilis*, automáticamente descargaba la información, le realizaba el BLAST, mostraba los resultados más favorables, e introducía la reacción al modelo. La descripción detallada está en Anexo 8.2. Cabe mencionar que estas vías abarcan los genes del metabolismo principal del HST28. Así, otros genes, tales como los encargados de la producción de metabolitos secundarios, son obviados para el desarrollo de este trabajo.

El proceso anteriormente descrito fue realizado 37 veces, una vez para cada mapa metabólico seleccionado de la base de datos de KEGG, tomando como referencia las versiones de estos mapas que eran específicas para *S. avermitilis* (código inicial "sma"). Los mapas utilizados están señalados en la Tabla 1. Para cada uno de ellos, se descargó la información de éste en formato KGML para procesar la información de las reacciones que se conocen presentes en *S. avermitilis* y que podrían estar en el HST28. Adicionalmente, se generó una planilla de formato .xlsx que hace de nexo entre el KGML y el código de ingreso de reacciones englobado en `RxnInter`. El desarrollo de esta planilla está mediado por un código, explicado en el Anexo 8.3

Tabla 1. Código y descripción de los mapas de KEGG (formato KGML) usados como base en la elaboración del modelo metabólico de HST28.

Código	Descripción
sma00010	Glicólisis y Gluconeogénesis.
sma00020	Ciclo TCA.
sma00030	Vía pentosa fosfato.
sma00040	Interconversiones de pentosa y glucuronato.
sma00051	Metabolismo de fructosa y manosa.
sma00052	Metabolismo de galactosa.
sma00061	Biosíntesis de ácidos grasos.
sma00071	Degradación de ácidos grasos.
sma00130	Biosíntesis de ubiquinona y otros terpenoides.
sma00220	Biosíntesis de arginina.
sma00230	Metabolismo de purinas.
sma00240	Metabolismo de pirimidinas.
sma00250	Metabolismo de alanina, aspartato y glutamato.
sma00260	Metabolismo de glicina, serina y treonina.

sma00270	Metabolismo de cisteína y metionina.
sma00290	Biosíntesis de valina, leucina e isoleucina.
sma00300	Biosíntesis de lisina.
sma00330	Metabolismo de arginina y prolina
sma00340	Metabolismo de histidina.
sma00400	Biosíntesis de fenilalanina, tirosina y triptófano.
sma00471	Metabolismo de D-glutamina y D-glutamato.
sma00500	Metabolismo de sacarosa y almidón.
sma00520	Metabolismo de aminoazúcares y azúcares de nucleótidos.
sma00563	Metabolismo de inositol fosfato.
sma00564	Metabolismo de glicerofosfolípidos.
sma00620	Metabolismo de piruvato.
sma00630	Metabolismo de glioxilato y dicarboxilato.
sma00640	Metabolismo de propanoato.
sma00650	Metabolismo de butanoato.
sma00660	Metabolismo de ácidos C5 ramificados.
sma00670	"One carbon pool by folate".
sma00740	Metabolismo de riboflavina.
sma00760	Metabolismo de nicotinato y nicotinamida.
sma00770	Metabolismo de pantotenato y CoA.
sma00790	Biosíntesis de folato.
sma00900	Biosíntesis de terpenoides.
sma00920	Metabolismo de azufre.

3.2.2 Adición de reacciones globales de biomasa

Las reacciones que llevan a la formación de biomasa, en vez de representar el cambio de un metabolito a otro como lo hacen las reacciones de conversión metabólica, representan la cantidad de masa de proteínas, DNA, RNA, carbohidratos y otros, que componen el peso total del organismo en estudio. Para añadirlas al sistema, se toman datos bibliográficos respecto a las proporciones en peso de cada componente, en base a organismos relacionados filogenéticamente.

Estas reacciones no tienen equivalente alguno en KEGG, y por ende, no se pueden utilizar los códigos utilizados para agregar las reacciones de conversión metabólica. Por tal razón, estas reacciones fueron agregadas manualmente, utilizando directamente las directrices sugeridas por COBRApy en su manual online ("Documentation for COBRApy — cobra 0.13.3 documentation," n.d.).

Las proporciones utilizadas para generar la ecuación de biomasa y sus componentes, según (Toro, Pinilla, Avignone, Rigoberto, & Estepa, 2018), se pueden ver en el Anexo 8.4.

3.2.3 Adición de reacciones de transporte e intercambio

Las reacciones de transporte son las encargadas de llevar metabolitos entre los espacios extra e intracelular, y están basados en propiedades reales de la estructura de la célula, como los canales difusión facilitada, las proteínas de transporte activo, entre otras. Por su parte, las reacciones de intercambio hacen referencia a la interacción entre el medio extracelular inmediato a la célula con el resto del espacio, representando las salidas y entrada totales del sistema. Se escriben de la manera "metabolito ⇔ ", indicando que por estas reacciones los componentes entran o salen del foco del ambiente celular.

Con el fin de agregar las reacciones de transporte e intercambio requeridas por el modelo, se desarrolló una función de Python capaz de agilizar el proceso (Ver Anexo 8.5). Para determinar qué reacciones agregar en este ítem, se basó en información hallada en KEGG, o bien en base a modelos previos hallados por bibliografía para *S. clavuligerus* (Toro et al., 2018) y para *S. coelicolor* (Borodina, Krabben, & Nielsen, 2005).

3.3 Curación manual del modelo

Para que el modelo funcionara, fue necesario hacerle diversos ajustes, principalmente ligados a metabolitos que aparecían en una sola reacción (vale decir, que no eran generados por ninguna especie, o que no derivaban a ninguna otra especie); o metabolitos que aparecían en muchas reacciones pero siempre con el mismo rol de reactante o producto. En base a lo hallado, se eliminaron algunas vías, y se agregaron otras que se veían estrictamente necesarias pero que no aparecían ligadas a ningún gen. Además, se agregaron las vías de transporte e intercambio para metabolitos seleccionados, en base a Para encontrar aquellos puntos que evitaban que el modelo diera resultados, vale decir, razones por la cual la optimización del modelo con vistas a la biomasa no entregaba valores, se ocupó un método de rastreo de errores basado en agregar salidas ficticias al sistema para determinar la dependencia del sistema de tal metabolito o reacción. El detalle del protocolo de rastreo se puede ver en el Anexo 8.6.

4. Resultados

4.1 Detalles del modelo

El modelo final cuenta con 804 reacciones, 627 metabolitos, y 594 genes. De tales reacciones, 83 corresponden a reacciones de transporte, 76 corresponden a reacciones de intercambio, y 11 son reacciones globales de formación de biomasa. De las 634 reacciones restantes, correspondientes a las conversiones metabólicas, 72 son gap, es decir, no tienen algún gen asociado. Ello equivale al 11,3% de las conversiones como gap.

De los metabolitos, 76 se encuentran en el medio externo. De los 551 que se encuentran en el citoplasma, 9 corresponden a macromoléculas usadas para determinar la biomasa.

En cuanto a la participación de los genes en cada proceso, se puede ver el resumen en la Figura 4. Destaca un alto porcentaje de genes destinados al metabolismo de nucleótidos y carbohidratos.

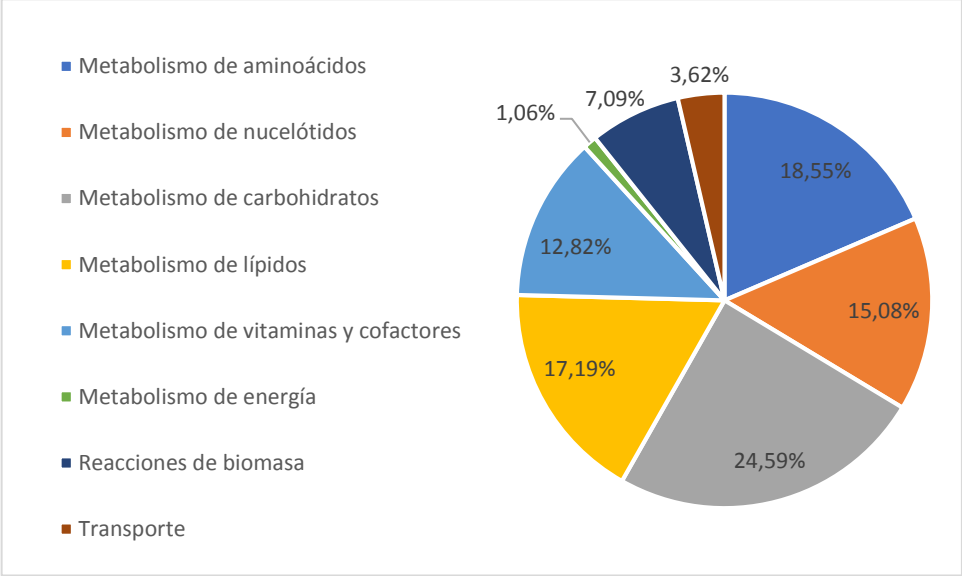

Figura 4. Participación del total de los genes del HST28 en cada proceso.

En un estado sin restricciones, vale decir, en un medio donde todos los metabolitos externos entran y salen libremente del sistema en cantidades libres, el principal consumo del modelo es el oxígeno y la glicina, seguido de la glucosa, representando entre los tres el 75% de la materia consumida por el modelo. Además, la principal fuente de nitrógeno en este estado sería la glutamina. En cuanto a la producción, el modelo estaría generando fundamentalmente ácido acético, con un 0,79% del total de masa producida dedicada a la formación de biomasa (ver Tabla 2).

Tabla 2. Consumo y producción de metabolitos del modelo sin restricciones. Porcentajes relativos al total de masa consumida y producida, respectivamente.

Consumo		Producción	
Metabolito	Porcentaje	Metabolito	Porcentaje
Glicina	27,04%	Biomasa	0,79%
Oxígeno	27,04%	Acetato	14,92%
D-Glucosa	25,98%	CO2	14,92%
Fumarato	9,27%	H+	14,92%
L-Glutamina	6,25%	Agua	14,92%
L-Serina	1,41%	L-Treonina	13,62%
L-Leucina	0,65%	Ortofosfato	9,71%
D-Fructosa	0,55%	L-Valina	7,25%
L-Arginina	0,51%	Amonio	4,60%
L-Prolina	0,39%	2-Oxoglutarato	4,24%
L-Fenilalanina	0,33%	Formato	0,07%
L-Lisina	0,16%	D-Alanina	0,04%
L-Histidina	0,15%		
L-Metionina	0,12%		
L-Triptófano	0,10%		
L-Cisteína	0,06%		

4.2 Comportamiento en medio mínimo

Al limitar las entradas del sistema a agua, oxígeno, una fuente de carbono (glucosa), una fuente de nitrógeno (amonio), una fuente de azufre (sulfato), y una fuente de fósforo (ortofosfato), se obtuvo un valor de producción de biomasa de 0,502 [h^{-1}], manteniendo una producción mayor de ácido acético, con 16,2 $\left[\frac{mmol}{gh}\right]$. Particularmente, los resultados indican un balance negativo de ortofosfato, siendo este producido en vez de consumido (ver Tabla 3).

Tabla 3. Consumo y producción de metabolitos en un medio con ingresos limitados.

Consumo		Producción	
Metabolito	Cantidad [mmol g^{-1} h^{-1}]	Metabolito	Cantidad [mmol g^{-1} h^{-1}]
Agua	12,00	Biomasa	0,502 [h^{-1}]
D-Glucosa	10,00	Ortofosfato	26,7
Amonio	4,43	Acetato	16,2
Oxígeno	4,42	H+	10,2
Sulfato	0,061	CO2	6,27
		D-Alanina	0,0453
		Formato	0,0238

4.3 Comportamiento ante fuentes de carbono

Al mantener el medio mínimo e ir variando las fuentes de carbono a ingresar al modelo, se obtuvo una mayor producción de biomasa al utilizar lactosa, maltosa, melibiosa, y especialmente sacarosa, obteniendo valores de hasta el 180% de la producción respecto a la glucosa. Hubo 9 fuentes que entregaron crecimiento cero o cercano a cero. En la Tabla 4 aparecen los valores, estando en negrita aquellos con producciones mayores a la glucosa.

Tabla 4. Producción de biomasa con diversas fuentes de carbono, normalizado con respecto al uso de glucosa.

Metabolito	Biomasa [h^{-1}]	Valor normalizado
D-Glucosa	0,5023	1,0000
(S)-Lactato	0,0000	0,0000
(S)-Malato	0,0352	0,0700
2-Oxoglutarato	0,1641	0,3268
Acetato	0,0000	0,0000
Ácido D-Gluconico	0,3349	0,6667
D-Fructosa	0,4034	0,8031
D-Gliceraldehído	0,1953	0,3889
D-Glicerato	0,0881	0,1754
D-Manosa	0,0000	0,0000
D-Sorbitol	0,0000	0,0000
Formato	0,0000	0,0000
Fumarato	0,0352	0,0700
Glicerol	0,0000	0,0000
Lactosa	**0,7059**	**1,4054**
Maltosa	**0,9041**	**1,8000**
Melibiosa	**0,9041**	**1,8000**
Piruvato	0,0881	0,1754
Sacarosa	**0,9076**	**1,8069**
Salicil Alcohol	0,0000	0,0000
Salicina	0,4018	0,8000
Shikimato	0,0000	0,0000
Succinato	0,0000	0,0000

4.4 Comportamiento ante fuentes de nitrógeno

Al sustituir el amonio por las fuentes de nitrógeno encontradas para el modelo, se observó un comportamiento idéntico en cuanto a la producción de biomasa, a excepción del 4-aminobutanoato, que registró una generación del 120% en comparación al amonio. En la Tabla 5 se ven estas diferencias, estando en negrita el que registra un cambio positivo respecto a la condición inicial.

Tabla 5. Producción de biomasa con diversas fuentes de nitrógeno, normalizado con respecto al uso de amonio.

Metabolito	Biomasa [h^{-1}]	Valor normalizado
Amonio	0,502	1
4-Aminobutanoato	**0,6028**	**1,2**
Nitrito	0,502	1
Urea	0,502	1

4.5 Uso de aminoácidos

Al usar aminoácidos como fuente de carbono y nitrógeno a una tasa de 0,1 $\left[\frac{mmol}{gh}\right]$, se encontró que el modelo puede utilizar 9 de estos 20 metabolitos como fuente exclusiva, aunque con crecimientos bajos (ver Tabla 6). El aminoácido que genera la mayor producción de biomasa, glutamina, fue analizado para ver los flujos de entrada y salida (Tabla 7). Se encontró que, a diferencia de lo visto en el medio mínimo, el modelo sí consume ortofosfato. Además, sigue produciendo acetato en cantidades considerablemente mayores que la producción de biomasa.

Tabla 6. Producción de biomasa al utilizar un único aminoácido como fuente de carbono y nitrógeno.

Aminoácido	Biomasa [h^{-1}] $x10^{-4}$
L-Alanina	9,767
L-Arginina	0,000
L-Asparagina	9,931
L-Aspartato	9,767
L-Cisteína	0,000
L-Fenilalanina	0,000
Glicina	4,684
L-Glutamato	9,767
L-Glutamina	19,86
L-Histidina	0,000
L-Isoleucina	0,000
L-Leucina	0,000
L-Lisina	0,000
L-Metionina	0,000
L-Prolina	19,53
L-Serina	11,07
L-Tirosina	0,000
L-Treonina	8,469
L-Triptófano	0,000
L-Valina	0,000

Tabla 7. Consumo y producción del modelo en un medio con glutamina como única fuente de carbono y nitrógeno.

Consumo		Producción	
Metabolito	Cantidad [mmol g^{-1} h^{-1}]	Metabolito	Cantidad [mmol g^{-1} h^{-1}]
Agua	$1{,}34x10^{-1}$	Biomasa	$1{,}99x10^{-3}$ [h^{-1}]
L-Glutamina	$1{,}00x10^{-1}$	Amonio	$1{,}82x10^{-1}$
Oxígeno	$4{,}82x10^{-2}$	Acetato	$1{,}68x10^{-1}$
Ortofosfato	$1{,}84x10^{-3}$	CO2	$3{,}21x10^{-2}$
Sulfato	$2{,}41x10^{-4}$	D-Alanina	$1{,}79x10^{-4}$
		Formato	$9{,}41x10^{-5}$

5. Discusiones

Respecto a las dimensiones del modelo, se puede decir que éste corresponde a uno de tamaño medio, en cuanto los modelos de escala genómica suelen tener 800 reacciones, 600 metabolitos y 650 genes en promedio (Oberhardt et al., 2009). Si se piensa que el genoma completo del HST28 cuenta con alrededor de ocho mil secuencias codificantes, un modelo con poco menos de 600 genes podría ser pensado como poco realista, especialmente si se cuenta que los modelos metabólicos promedian una cobertura del 22% de los genes del organismo aproximadamente (Cortés et al., 2017), mientras que este modelo cubre el 7,5% de las secuencias codificantes del HST28.

Es necesario destacar, sin embargo, que este modelo se centró en lo que se consideró necesario para la subsistencia de la célula, más que en la producción de metabolitos adicionales por parte de ésta. Es válido pensar que, de agregar las vías metabólicas para compuestos especializados, los números de metabolitos, reacciones y genes presentes en el modelo crecerían considerablemente, por lo que el tamaño del modelo no debería ser un motivo de cuestionamiento.

En cuanto a la presencia de reacciones sin genes asociados en el modelo (gaps), el 11,3% registrado en este modelo es un valor prudente en comparación a otros modelos, que promedian el 20% (Borodina et al., 2005)(Cortés et al., 2017)(Razmilic, Castro, Andrews, & Asenjo, 2018). Si bien asumir la presencia de reacciones en el metabolismo de la célula en base a un estado ideal puede sonar antojadizo, es una práctica común que ha dado buenos resultados, por lo que tener gaps en un modelo no es dañino. Y si a ello se le suma una cantidad baja de gaps, se puede afirmar que el modelo está bien constituido en ese aspecto.

En cuanto al desempeño del modelo en condiciones irrestrictas, llama la atención el alto consumo de glicina registrado, siendo mayor que el de glucosa y estando a la par con el de oxígeno. Este comportamiento no se condice con lo registrado en el uso de aminoácidos, en cuanto se esperaría que el mayor ingreso del sistema lo tenga aquel que más biomasa produce por sí solo, que es la glutamina. Una explicación para este comportamiento podría radicar en el hecho de la existencia de más de un óptimo, vale decir, más de una configuración de flujos que llevan al mismo resultado. Ello implica que podría ser recomendable repetir las optimizaciones para obtener un espectro más amplio de información acerca de lo que puede estar pasando en el modelo.

Al observar el desempeño del modelo frente a un ambiente con recursos limitados, destaca la producción de ortofosfato como una anomalía. No tiene sentido que salga fósforo del sistema siendo que éste no entra. Una

explicación para este fenómeno podría ser la presencia de bucles (loops) en el modelo. Éstos loops son cadenas de reacciones en las cuales hay uno o más metabolitos que son generados y consumidos de manera cíclica. Particularmente, es probable que haya ciclos de generación y consumo de ortofosfato al ser éste un metabolito común presente en varias reacciones del modelo.

Respecto al uso de fuentes de carbono, se puede apreciar que aquellos metabolitos que generaron una mayor obtención de biomasa son todos disacáridos. Una explicación somera sería decir que tales metabolitos entregan el doble de monosacáridos que la glucosa, y que por ende alcanzan rendimientos mayores. Sin embargo, este razonamiento no es del todo cierto, en cuanto los valores alcanzados por los disacáridos suelen ser menores que la suma de los dos monosacáridos por separado. Para visualizar ello, se presenta la Tabla 8.

Tabla 8. Comparación de producción de biomasa entre disacáridos y sus monosacáridos respectivos como única fuente de carbono. (r) indica que la entrada de tal metabolito sí está en el modelo. (f) indica que la entrada es ficticia, válida exclusivamente para efectos de esta tabla.

Metabolito	Productos	Biomasa [h^{-1}]
Lactosa	D-Galactosa α-D-Glucosa	0,7059
Maltosa	D-Glucosa D-Glucosa	0,9041
Melibiosa	D-Glucosa D-Galactosa	0,9041
Sacarosa	D-Glucosa D-Fructosa	0,9076

Metabolito	Biomasa [h^{-1}]
α-D-Glucosa (f)	0,403
D-Fructosa (r)	0,4034
D-Galactosa (f)	0,402
D-Glucosa (r)	0,5023

Si bien la lactosa y la melibiosa están conformadas por glucosa y galactosa, el hecho de que el modelo distinga entre α-D-glucosa y D-glucosa hace que la glucosa de la lactosa sea menos accesible para el modelo. Por otra parte, la maltosa y la melibiosa alcanzan el mismo valor principalmente porque la vía metabólica que comienza con la galactosa desemboca en la D-glucosa-1P, al igual que la vía que empieza con la entrada de glucosa. Por ende, al unirse las rutas, es esperable que confluyan al mismo valor de biomasa.

Respecto al uso de fuentes de nitrógeno, el modelo parece poder usar indistintamente el amonio, el nitrito y la urea. En el caso de esta última, el modelo la transforma a amonio como primer paso para su uso, lo cual explica que lleven al mismo resultado. En cuanto al nitrito, ocurre el mismo

fenómeno: Se transforma totalmente en amonio después de entrar al sistema.

En cuanto al uso de aminoácidos, queda en evidencia cuáles son aquellos que el modelo puede metabolizar para realizar todos los procesos celulares pertinentes. También se observan las limitaciones del metabolismo en el modelo, siendo éste incapaz de procesar 11 de los 20 aminoácidos como fuente única de nitrógeno. Cabe destacar la glutamina como el mejor sustrato aminoacídico para el modelo, según los resultados vistos.

A modo general del resultado del modelo, hay dos aspectos a destacar. El primero es la alta producción de ácido acético presente en todas las simulaciones del modelo. Según bibliografía ("Streptomyces coelicolor A3(2) Pathway: acetate formation from acetyl-CoA I," n.d.), es frecuente la excreción de acetato en bacterias con el fin de mantener en el ambiente una fuente de carbono para utilizar una vez que se acaben otras fuentes, como la glucosa. Sin embargo, la simulación no presentó consumo de acetato cuando se dejó a esta molécula como única fuente de carbono. Esta situación da pie a una posible recomendación de manejar con cuidado los niveles de acetato en caso de intentar cultivar la bacteria.

El otro aspecto a destacar está relacionado a los valores registrados para la producción de biomasa en todos los casos. Para la regla general, los valores resultantes de este trabajo son demasiado altos, considerando que un crecimiento de 0,5 [h^{-1}] ya se considera elevado. Si bien no es descartable del todo la idea de tener valores mayores al promedio a la hora de hacer crecer la bacteria, es plausible pensar en algún tema de fondo que pueda estar generando tales valores. Particularmente, es posible que los loops discutidos previamente estén alterando los valores de flujo, al generar estructuras cíclicas en las cuales los flujos viajan indefinidamente. Ante esta situación, se puede dejar por claro que estos flujos están sobreestimados, a la hora de comparar con una eventual validación del modelo.

Continuando con la última línea del párrafo anterior, es importante el poder validar el modelo mediante métodos tradicionales de cultivo. Es crucial el poder anexar el mundo teórico computacional con la parte práctica. Sin embargo, es necesario advertir que la calidad del genoma secuenciado no era de buen estándar, por lo que puede haber discrepancias entre lo generado *in silico* y lo observable *in vivo*. Es también importante mencionar que este mismo punto de la calidad dificultaría el buscar rutas de metabolitos secundarios, en cuanto se requiere una precisión mayor para identificar vías no estudiadas en tiempos anteriores.

Más allá del resultado del modelo en sí, es importante destacar el proceso que se realizó para generar éste. Se logró desarrollar un protocolo que permite obtener toda la potencialidad de la base de datos de KEGG y llevarla al uso para elaborar modelos metabólicos. La serie de códigos permite a un usuario con conocimientos promedio de Python el hacer su propio modelo

metabólico con complicaciones menores. El protocolo podría ser usado por futuros investigadores para armar sus modelos de manera más rápida que con otros métodos, teniendo así este trabajo realizado un potencial de contribuir más allá que tan solo con un modelo, si no que con muchos modelos más. En base a lo expuesto en este párrafo, es posible afirmar que el proceso puede tener más potencial que el resultado a la hora de contribuir con futuras investigaciones.

6. Conclusiones

La principal observación del presente trabajo es la elaboración del modelo requerido del HST28, con dimensiones aceptables según los promedios de tamaño encontrados. Si bien la representación de regiones codificantes del genoma completo en los genes del modelo es baja, no afecta la calidad del resultado final. En cuanto al número de gaps, el modelo también cumple entre los estándares conocidos, justificando en cierto grado su existencia y demostrando que el criterio a la hora de agregar gaps no estuvo fuera de los márgenes razonables.

En cuanto a las condiciones de crecimiento, se sugiere que, en un eventual intento de cultivo, se administre sacarosa al medio como fuente de carbono. En cuanto a fuentes de nitrógeno, se podría usar indistintamente amonio, nitrito o urea, y si es factible, 4-aminobutanoato.

Si bien la validación del modelo escapa de los alcances de este trabajo, se recomienda realizar tal proceso en algún trabajo futuro, para poder comprobar de manera tangible las conclusiones llevadas a cabo en este informe.

A modo de sugerencia para trabajos futuros, se recomienda fuertemente una nueva secuenciación del genoma del HST28 para poder extraer mejores resultados a partir de éste. Adicionalmente, podría servir para sobreescribir elementos dudosos del modelo (ej: genes con bajas identidades) y ayudar a tener un sistema más robusto.

Se llevó a cabo el diseño de todo un sistema de elaboración de modelos metabólicos basado en KEGG. Si bien este no era un objetivo del trabajo en primera instancia, es un paso importante en cuanto a su utilidad, que se extiende a dimensiones que exceden el alcance de este trabajo. A ello se le agrega el protocolo para detectar errores, que también puede ser usado de manera ajena a este caso particular. Los códigos aquí desarrollados tienen la potencialidad de facilitar considerablemente el trabajo de otros investigadores del campo de la bioinformática, quedando quizás como el legado más patente que podría dejar este trabajo de tesis.

En resumen, se cumplieron los objetivos planteados originalmente, y se obtuvieron resultados adicionales que pueden resultar de alto valor.

7. Bibliografía

Bakker, M., van der Graaf, W. T. A., Groen, H. J. ., Smit, E. F., & De Vries, E. G. E. (1995). Anthracyclines - Pharmacology and Resistance, A Review. *Current Pharmaceutical Design*, 133–144.

Becker, S. A., Feist, A. M., Mo, M. L., Hannum, G., Palsson, B. Ø., & Herrgard, M. J. (2007). Quantitative prediction of cellular metabolism with constraint-based models: the COBRA Toolbox. https://doi.org/10.1038/nprot.2007.99

Bentley, S. D., Chater, K. F., Cerdeñ O-Tá Rraga, A.-M., Challis, G. L., Thomson, N. R., James, K. D., … Hopwood, D. A. (2002). *Complete genome sequence of the model actinomycete Streptomyces coelicolor A3(2)*. Retrieved from www.ebi.ac.uk/genomes/

BLAST® Command Line Applications User Manual. (n.d.). Retrieved March 3, 2019, from https://www.ncbi.nlm.nih.gov/books/NBK279670/

Borodina, I., Krabben, P., & Nielsen, J. (2005). Genome-scale analysis of Streptomyces coelicolor A3(2) metabolism. *Genome Research*, *3*(2), 820–829. https://doi.org/10.1101/gr.3364705.metabolism

Bull, A. T., & Asenjo, J. A. (2013). Microbiology of hyper-arid environments: Recent insights from the Atacama Desert, Chile. *Antonie van Leeuwenhoek, International Journal of General and Molecular Microbiology*, *103*(6), 1173–1179. https://doi.org/10.1007/s10482-013-9911-7

Cokelaer, T., Pultz, D., Harder, L. M., Serra-Musach, J., Saez-Rodriguez, J., & Valencia, A. (2013). BioServices: A common Python package to access biological Web Services programmatically. *Bioinformatics*, *29*(24), 3241–3242. https://doi.org/10.1093/bioinformatics/btt547

Cortés, M. P., Mendoza, S. N., Travisany, D., Gaete, A., Siegel, A., Cambiazo, V., & Maass, A. (2017). Analysis of Piscirickettsia salmonis metabolism using genome-scale reconstruction, modeling, and testing. *Frontiers in Microbiology*, *8*(DEC). https://doi.org/10.3389/fmicb.2017.02462

Documentation for COBRApy — cobra 0.13.3 documentation. (n.d.). Retrieved April 26, 2019, from https://cobrapy.readthedocs.io/en/latest/

Dorador, C., Vila, I., Imhoff, J. F., & Witzel, K. (2008). Cyanobacterial diversity in Salar de Huasco , a high altitude saline wetland in northern Chile : an example of geographical dispersion ?, *1*. https://doi.org/10.1111/j.1574-6941.2008.00483.x

Dorador, C., Vila, I., Remonsellez, F., Imhoff, J. F., & Witzel, K. (2010). Unique clusters of Archaea in Salar de Huasco , an athalassohaline evaporitic basin of the Chilean Altiplano. https://doi.org/10.1111/j.1574-6941.2010.00891.x

Ebrahim, A., Lerman, J. A., Palsson, B. O., & Hyduke, D. R. (2013). COBRApy: COnstraints-Based Reconstruction and Analysis for Python. *BMC Systems Biology*, *7*. https://doi.org/10.1186/1752-0509-7-74

Ehrlich, J., Bartz, Q. R., Smith, R. M., Joslyn, D. A., & Burkholder, P. R. (1947). Chloromycetin, a New Antibiotic From a Soil Actinomycete. *Science*.

https://doi.org/10.1126/science.106.2757.417

Hover, B. M., Kim, S.-H., Katz, M., Charlop-Powers, Z., Owen, J. G., Ternei, M. A., … Brady, S. F. (2018). Culture-independent discovery of the malacidins as calcium-dependent antibiotics with activity against multidrug-resistant Gram-positive pathogens. https://doi.org/10.1038/s41564-018-0110-1

Ingraham, J. (1983). *Growth of the Bacterial Cell*. Sinauer Associates.

Kanehisa. (2019a). Current Statistics. Retrieved April 19, 2019, from https://www.genome.jp/kegg/docs/statistics.html

Kanehisa. (2019b). KEGG Overview. Retrieved April 19, 2019, from https://www.genome.jp/kegg/kegg1a.html

King, Z. A., Dräger, A., Ebrahim, A., Sonnenschein, N., Lewis, N. E., & Palsson, B. O. (2015). Escher: A Web Application for Building, Sharing, and Embedding Data-Rich Visualizations of Biological Pathways. *PLoS Computational Biology*, *11*(8), 1–13. https://doi.org/10.1371/journal.pcbi.1004321

Lacap, D. C., Warren-Rhodes, K. A., McKay, C. P., & Pointing, S. B. (2011). Cyanobacteria and chloroflexi-dominated hypolithic colonization of quartz at the hyper-arid core of the Atacama Desert, Chile. *Extremophiles*, *15*(1), 31–38. https://doi.org/10.1007/s00792-010-0334-3

Laxminarayan, R., Duse, A., Wattal, C., Zaidi, A. K. M., Wertheim, H. F. L., Sumpradit, N., … Cars, O. (2013). Antibiotic resistance-the need for global solutions. *The Lancet Infectious Diseases*, *13*(12), 1057–1098. https://doi.org/10.1016/S1473-3099(13)70318-9

Lomovskaya, N., Otten, S. L., Doi-Katayama, Y., Fonstein, L., Liu, X. C., Takatsu, T., … Hutchinson, C. R. (1999). Doxorubicin overproduction in Streptomyces peucetius: Cloning and characterization of the dnrU ketoreductase and dnrV genes and the doxA cytochrome P-450 hydroxylase gene. *Journal of Bacteriology*, *181*(1), 305–318.

Madigan, M. T. (2005). Brock Biology of Microorganisms, 11th edn. *International Microbiology*.

Oberhardt, M. A., Palsson, B. Ø., & Papin, J. A. (2009). Applications of genome-scale metabolic reconstructions. *Molecular Systems Biology*, *5*(1), 320. https://doi.org/10.1038/MSB.2009.77

Pantel, L., Florin, T., Dobosz-Bartoszek, M., Racine, E., Sarciaux, M., Serri, M., … Gualtieri, M. (2018). Odilorhabdins, Antibacterial Agents that Cause Miscoding by Binding at a New Ribosomal Site. *Molecular Cell*. https://doi.org/10.1016/j.molcel.2018.03.001

Piddock, L. J. V. (2015). Teixobactin, the first of a new class of antibiotics discovered by ichip technology? *Journal of Antimicrobial Chemotherapy*, *70*(10), 2679–2680. https://doi.org/10.1093/jac/dkv175

Razmilic, V., Castro, J. F., Andrews, B., & Asenjo, J. A. (2018). Analysis of metabolic networks of Streptomyces leeuwenhoekii C34 by means of a genome scale model : Prediction of modifications that enhance the production of specialized metabolites. *Biotechnology and Bioengineering*, *115*(October 2017), 1815–1828. https://doi.org/10.1002/bit.26598

Risacher, F. (1999). GEOQUIMICA DE AGUAS EN CUENCAS CERRADAS: I, II Y III REGIONES - CHILE.

Schatz, A., Bugie, E., & Waksman, S. A. (1944). Streptomycin, a Substance Exhibiting Antibiotic Activity Against Gram- Positive and Gram-Negative Bacteria. *Clinical Orthopaedics and Related Research*. https://doi.org/10.3181/00379727-55-14461

Schellenberger, J., Que, R., Fleming, R. M. T., Thiele, I., Orth, J. D., Feist, A. M., … Palsson, B. (2011). Quantitative prediction of cellular metabolism with constraint-based models: The COBRA Toolbox v2.0. *Nature Protocols*, *6*(9), 1290–1307. https://doi.org/10.1038/nprot.2011.308

Schulz, D., Beese, P., Ohlendorf, B., Erhard, A., Zinecker, H., Dorador, C., & Imhoff, J. F. (2011). Abenquines A – D : aminoquinone derivatives produced by Streptomyces sp . strain DB634, *64*(12), 763–768. https://doi.org/10.1038/ja.2011.87

Sielfield, W., Amado, N., Herreros, J., & Peredo, R. (1996). La avifauna del Salar del Huasco: Primera región, Chile. *Boletín Chileno de Ornitología*, (3), 17–24.

Stach, J. E. M., & Bull, A. T. (2005). Estimating and comparing the diversity of marine actinobacteria, 3–9. https://doi.org/10.1007/s10482-004-6524-1

Streptomyces coelicolor A3(2) Pathway: acetate formation from acetyl-CoA I. (n.d.). Retrieved May 1, 2019, from https://biocyc.org/SCO/NEW-IMAGE?type=PATHWAY&object=PWY0-1312

Thomas, D. (1997). Arid zones: their nature and extent. In *Arid zone geomorphology* (2nd ed., pp. 3–12). Chichester: Wiley.

Toro, L., Pinilla, L., Avignone, C., Rigoberto, R., & Estepa, R. (2018). An enhanced genome-scale metabolic reconstruction of Streptomyces clavuligerus identifies novel strain improvement strategies. *Bioprocess and Biosystems Engineering*, *41*(5), 657–669. https://doi.org/10.1007/s00449-018-1900-9

UMEZAWA, H., UEDA, M., MAEDA, K., YAGISHITA, K., KONDO, S., OKAMI, Y., … TAKEUCHI, T. (1957). Production and isolation of a new antibiotic: kanamycin. *The Journal of Antibiotics*. https://doi.org/10.1093/nar/gkt1209

Waksman, S. A., & Lechevalier, H. A. (1949). Neomycin, a new antibiotic active against streptomycin-resistant bacteria, including tuberculosis organisms. *Science*. https://doi.org/10.1126/science.109.2830.305

Yoon, S., Ha, S., Kwon, S., Lim, J., Kim, Y., Seo, H., & Chun, J. (2019). Introducing EzBioCloud : a taxonomically united database of 16S rRNA gene sequences and whole-genome assemblies, 1613–1617. https://doi.org/10.1099/ijsem.0.001755

Zaitlin, B., & Watson, S. B. (2006). Actinomycetes in relation to taste and odour in drinking water: Myths, tenets and truths. *Water Research*. https://doi.org/10.1016/j.watres.2006.02.024

Zengler, K., Paradkar, A., & Keller, M. (2004). *2 New Methods to Access Microbial Diversity for Small Molecule Discovery*. Retrieved from https://link-springer-com.uchile.idm.oclc.org/content/pdf/10.1007%2F978-1-59259-976-9_12.pdf

Zenova, G. M., Manucharova, N. A., & Zvyagintsev, D. G. (2014). Extremophilic and Extremotolerant Actinomycetes, (April 2011). https://doi.org/10.1134/S1064229311040132

8. Anexos

8.1 Resultados de análisis de identificación EzBioCloud

Tabla 9. Identificación por análisis de RNA 16S.

Rank	Name	Strain	Authors	Pairwise Similarity(%)
1	Streptomyces kanamyceticus	NBRC 13414	Okami and Umezawa 1957	99,0318119
2	CP022685_s	KY5		98,8950276
3	Streptomyces aureus	NBRC 100912	Manfio et al. 2003	98,8950276
4	Streptomyces durmitorensis	MS405	Savic et al. 2007	98,8934993
5	Streptomyces longisporoflavus	NBRC 12886	Waksman 1953	98,7252125
6	Streptomyces rosealbus	YIM 31634	Xu et al. 2012	98,5590778
7	Streptomyces olivochromogenes	DSM 40451	(Waksman 1923) Waksman and Henrici 1948	98,5497238
8	Streptomyces resistomycificus	NRRL ISP-5133	Lindenbein 1952	98,3425414
9	Streptomyces setonii	NRRL ISP-5322	(Millard and Burr 1926) Waksman 1953	98,2044199
10	Streptomyces mirabilis	NBRC 13450	Ruschmann 1952	98,203179
11	Streptomyces yanii	NBRC 14669	Liu et al. 2005	98,1884058
12	MJAH_s	LUP47b		98,1353591
13	Streptomyces canus	DSM 40017	Heinemann et al. 1953	98,1353591
14	Streptomyces badius	NRRL B-2567	(Kudrina 1957) Pridham et al. 1958	98,1353591
15	Streptomyces rhizosphaerihabitans	JR-35	Lee and Whang 2016	98,1340705
16	Streptomyces parvus	NBRC 3388	(Krainsky 1914) Waksman and Henrici 1948	98,1327801
17	Streptomyces sindenensis	NBRC 3399	Nakazawa and Fujii 1957	98,1327801
18	Streptomyces arcticus	ZLN234	Zhang et al. 2016	98,0662983
19	Streptomyces phaeoluteigriseus	DSM 41896	Goodfellow et al. 2008	98,0662983
20	Streptomyces griseorubiginosus	DSM 40469	(Ryabova and Preobrazhenskaya 1957) Pridham et al. 1958	97,9972376
21	Streptomyces anulatus	NRRL B-2000	(Beijerinck 1912) Waksman 1953	97,9972376
22	Streptomyces variegatus	NRRL B-16380	Sveshnikova and Timuk 1986	97,9972376

23	Streptomyces flavovariabilis	NRRL B-16367	(ex Korenyako and Nikitina) Sveshnikova 1986	97,9972376
24	Streptomyces bottropensis	ATCC 25435	Waksman 1961	97,9972376
25	Streptomyces phaeochromogenes	NBRC 3180	(Conn 1917) Waksman 1957	97,9958535
26	Streptomyces cinereoruber subsp. fructofermentans	NBRC 15396	Corbaz et al. 1957	97,9875087
27	Streptomyces roseolilacinus	NBRC 12815	(Preobrazhenskaya and Sveshnikova 1957) Pridham et al. 1958	97,9861111
28	Streptomyces fulvissimus	DSM 40593	(Jensen 1930) Waksman and Henrici 1948	97,9281768
29	Streptomyces avermitilis	MA-4680	(ex Burg et al. 1979) Kim and Goodfellow 2002	97,9281768
30	Streptomyces luridiscabiei	NRRL B-24455	Park et al. 2003	97,9281768
31	Streptomyces microflavus	NBRC 13062	(Krainsky 1914) Waksman and Henrici 1948	97,926745
32	Streptomyces globisporus	NBRC 12867	(Krassilnikov 1941) Waksman 1953	97,9253112
33	Streptomyces fulvorobeus	NBRC 15897	Vinogradova and Preobrazhenskaya 1986	97,9238754
34	Streptomyces pluricolorescens	NBRC 12808	Okami and Umezawa 1961	97,9209979
35	Streptomyces ziwulingensis	F22	Lin et al. 2013	97,9196557
36	Streptomyces rubiginosohelvolus	NBRC 12912	(Kudrina 1957) Pridham et al. 1958	97,9166667
37	Streptomyces ederensis	NBRC 15410	Wallhäusser et al. 1966	97,8917779
38	Streptomyces pratensis	ch24	Rong et al. 2014	97,8832117
39	Streptomyces rectiviolaceus	NRRL B-16374	(ex Artamonova) Sveshnikova 1986	97,859116
40	Streptomyces stelliscabiei	NRRL B-24447	Bouchek-Mechiche et al. 2000	97,859116
41	Streptomyces sanglieri	NBRC 100784	Manfio et al. 2003	97,859116
42	Streptomyces cinnabarigriseus	JS360	Landwehr et al. 2018	97,859116
43	Streptomyces flavovirens	NBRC 3716	(Waksman 1923) Waksman and Henrici 1948	97,859116
44	Streptomyces similanensis	KC-106	Sripreechasak et al. 2016	97,859116
45	Streptomyces cyaneofuscatus	NRRL B-2570	(Kudrina 1957) Pridham et al. 1958	97,859116

46	Streptomyces pseudovenezuelae	DSM 40212	(Kuchaeva et al. 1961) Pridham 1970	97,859116
47	Streptomyces bobili	NRRL B-1338	(Waksman and Curtis 1916) Waksman and Henrici 1948	97,859116
48	Streptomyces humidus	NBRC 12877	Nakazawa and Shibata 1956	97,8576365
49	Kitasatospora albolonga	NBRC 13465	(Tsukiura et al. 1964) Labeda et al. 2017	97,8561549
50	Streptomyces griseus subsp. griseus	KCTC 9080	(Krainsky 1914) Waksman and Henrici 1948	97,8561549

8.2 Descripción de código de desarrollo del modelo

El código de Python utilizado para generar el modelo cuenta con diversas funciones que operan interrelacionadas. A continuación, se describirán las operaciones desarrolladas en detalle. Para cada función, está el código en el Anexo 8.7, bajo el capítulo homónimo. Las marcas en corchetes ("[]") indican la línea del código respectivo al cual se hace referencia.

8.2.1 RxnInter

Tabla 10. Resumen de la función RxnInter.

Entradas		
Variable	Tipo	Significado
`ggroups`	Lista de listas de valores.	Genes de *S. avermitilis* participantes en el proceso. Cada sub-lista es una reacción distinta. Solo viene la parte numérica del nombre del gen.
`name`	String.	Identificador de la vía metabólica estudiada.
`modelo`	Objeto `model` de COBRAPy.	Modelo al cual se le agregan las reacciones.
`excel`	Planilla de cálculo.	Planilla con información de reacciones y cambios (optativo).
`exdata`	Lista con un string y un valor.	El string es el nombre de la hoja de la planilla. El valor es la primera fila de la hoja a leer o escribir. (optativo).
Salidas		
Esta función no genera salidas.		

Ésta es la función principal del método de adición de reacciones desarrollado. Con las tres primeras variables de la Tabla 10, el programa es capaz de agregar las reacciones deseadas al modelo metabólico. La función está

diseñada para obtener información específicamente del *S. avermitilis*, para la cual se ingresa como código la parte numérica del nombre del gen asociado (Los genes de *S. avermitilis* tienen como formato de nombre la estructura "SAVERM_####"). Las variables `excel` y `exdata` están relacionadas a la escritura de una planilla de cálculo con fines de presentar al usuario la información de manera ordenada. Cabe mencionar que, cambiando la línea de código que hace referencia a la parte alfabética del identificador de los genes [12], la función podría extender su alcance a otras bases de datos de genomas que incluyan formatos de nombre similares.

La función `RxnInter` genera internamente una lista con los nombres completos de los genes de *S. avermitilis*, omitiendo genes repetidos (ej: un gen que regula dos reacciones) [6 a 13]. Posteriormente, `RxnInter` llama a la función `FastaMaker` para generar un archivo FASTA con los genes requeridos (ver Anexo 8.2.2), y luego a la función `Blaster` para realizarle un BLAST local al archivo FASTA ya mencionado (ver Anexo 8.2.3).

Después de llamar a `Blaster`, la función `RxnInter` lee el archivo .xml recién generado, y lo subdivide mediante un método de Biopython diseñado para esta tarea (`NCBIXML.parse`). Posteriormente, se genera una lista con resultados a partir de la información subdividida (llamada `result`), la cual está separada internamente según cada gen de *S. avermitilis*, con sus respectivas coincidencias en el HST28 [16 a 18]. En este punto, se abre un ciclo `for` que itera sobre la lista `ggroups` definida anteriormente, de modo tal que se trabaja una reacción por cada iteración [22].

En cada iteración, el programa informa al usuario el número de la reacción sobre la cual se está trabajando, y luego itera nuevamente sobre cada gen presente en la reacción, para procesarlos mediante la función `HSTRetrieval`, que obtiene los genes del HST28 a partir de los de *S. avermitilis* (Anexo 8.2.4).

Habiendo pasado por la acción de `HSTRetrieval`, si `RxnInter` detecta un elemento `None`, le pregunta al usuario si desea continuar con el proceso para tal reacción, debiendo éste responder con un 1 (sí) o un 0 (no). En caso afirmativo, el programa entra a la función `Gene2Rxn` para continuar con el desarrollo del trabajo. Si la función no encuentra un `None`, el programa procede a `Gene2Rxn` automáticamente, sin intervención del usuario (Ver Anexo 8.2.5).

La función `RxnInter` pasará por `HSTRetrieval`, y cuando corresponda, por `Gene2Rxn`, para cada reacción estudiada, según las iteraciones del ciclo `for`. Una vez terminado el ciclo, la función le indica al usuario que la operación ha llegado a su fin mediante un string [45].

8.2.2 FastaMaker

Tabla 11. Resumen de la función FastaMaker.

Entradas		
Variable	Tipo	Significado
`genes`	Lista de strings.	Genes de *S. avermitilis* (nombre completo).
`name`	String.	Nombre del archivo .FASTA.
Salidas		
No tiene salidas al entorno del programa. Sin embargo, genera un archivo .FASTA a la carpeta en la cual se está trabajando.		

Esta función usa la lista de nombres completos de *S. avermitilis* en conjunto con el nombre entregado a `RxnInter` para generar un archivo .FASTA con la secuencia aminoacídica de todos los genes incluidos en la lista.

Particularmente, `FastaMaker` contiene una clase KEGG de Bioservices, mediante la cual se descarga la información requerida de cada gen, representada en KEGG mediante la secuencia de aminoácidos traducida a partir de los nucleótidos respectivos [2 a 6]. Las secuencias son almacenadas en una variable, la cual es posteriormente traspasada al archivo FASTA [9].

Como se ve en la Tabla 11, el programa no tiene una salida más allá del documento generado. El archivo FASTA queda disponible en la carpeta en la cual se ejecuta el código, para uso inmediato.

8.2.3 Blaster

Tabla 12. Resumen de la función Blaster.

Entradas		
Variable	Tipo	Significado
name	String.	Nombre del archivo FASTA a leer, y a la vez nombre del archivo a generar.
Salidas		
No tiene salidas al entorno del programa. Sin embargo, genera un archivo xml a la carpeta en la cual se está trabajando.		

La función `Blaster`, a partir de su único argumento de nombre (`name`, ver Tabla 12), lee el archivo FASTA generado por FastaMaker cuyo nombre coincide con el argumento , le realiza el BLAST a cada gen presente en el archivo usando el genoma del HST28 como base de datos, y entrega un nuevo archivo .xml del mismo nombre, en el cual se guardan los resultados

del BLAST para cada gen ingresado en el FASTA. Cabe mencionar que esta función está optimizada para trabajar con la HST28 y para emitir el resultado en el formato ideal para este trabajo (xml), utilizando el mismo nombre. Es posible agregar un par de parámetros más a la entrada de la función, modificando la línea 2 del código (Anexo 8.7.3) para responder a tales entradas, para volverla de uso general.

8.2.4 HSTRetrieval

Tabla 13. Resumen de la función HSTRetrieval.

Entradas		
Variable	Tipo	Significado
`gen`	String.	Gen de *S. avermitilis* al cual se le encontrará análogo en HST28.
`resultado`	Objeto `NCBIXML.parse` de Biopython.	Resultado del BLAST local sobre las secuencias a estudiar.
Adicionalmente, requiere entradas del tipo `input`, de carácter numérico.		
Salidas		
Variable	Tipo	Significado
`reserve[i]`	String.	Gen de HST28 que regula la reacción en estudio.
Esta función puede tener como salida un valor vacío (`None`).		

El funcionamiento de `HSTRetrieval` es interactivo, vale decir, requiere intervenciones del usuario para llevar a cabo su tarea. En primer lugar, la función toma los componentes de la variable `result` de `RxnInter` (llamada aquí `resultado` para evitar coincidencias de nombre) y busca entre los genes que fueron ingresados al BLAST a aquel que coincide con el código del gen requerido (`gen`). Una vez hecha la correlación entre ambos elementos, el programa muestra todos los genes de HST28 que la operación BLAST identificó para tal gen de *S. avermitilis*, que cumplen con los umbrales de identidad (`i_tres`, 45% o mayor) y de valor E (`e_tres`, 10^{-3} o menor) utilizados para indicar la calidad del resultado (Razmilic et al., 2018). El programa le muestra al usuario, para cada gen de HST28 identificado, el nombre de éste, el valor de identidad y el valor E, con el fin de que el usuario decida si quiere agregar este gen como responsable de la reacción trabajada [14 a 19]. Dependiendo de la cantidad de genes candidatos, el método de ingreso de la respuesta del usuario varía. Si se presenta solo un gen, el usuario debe escoger entre agregarlo o no agregarlo, lo cual se transmite al entorno virtual de manera binaria, debiendo ingresar el usuario un 1 (sí se agrega) o un 0 (no se agrega) [26]. Si hay más genes, el usuario debe ingresar el número asociado al gen que quiere agregar, o ingresar un 0 en caso de no querer agregar ninguno [34]. En caso de no haber coincidencias, el programa informa al usuario esta situación, y devuelve un

valor `None` como respuesta [23 y 24]. Si el usuario decide no agregar ninguno de los genes candidatos, la función también devuelve un `None`. Como figura en la Tabla 13, las posibles salidas están en la variable `reserve`, y el índice `i` representa el valor asignado por el usuario.

8.2.5 Gene2Rxn

Tabla 14. Resumen de la función Gene2Rxn.

Entradas		
Variable	Tipo	Significado
`grupo`	Lista de valores numéricos.	Genes de *S. avermitilis* involucrados en la reacción.
`HST`	Lista de strings.	Genes de HST28 involucrados en la reacción.
`j`	Valor numérico.	Índice de la reacción.
`modelo`	Objeto `model` de COBRAPy.	Modelo al cual se le agrega la reacción.
`xlsinfo`	Lista de un string y dos valores numéricos.	El string es el nombre de la hoja de la planilla de cálculo a usar. Los valores numéricos son indicadores de la posición de la fila a usar en la planilla (optativo).
Salidas		
Esta función no genera salidas.		

La función `Gene2Rxn` toma como argumento los parámetros señalados en la Tabla 14 para identificar, descargar y agregar al modelo la información respectiva a la reacción que se está estudiando. `Gene2Rxn`, usando la variable `grupo`, identifica en primer lugar la ortología asociada a los genes de *S. avermitilis*, asumiendo que tales genes corresponden probablemente a la misma ortología. En caso de encontrarse más de una ortología, se le expone el problema al usuario, con el fin de que éste decida con cuál de ellas se prosigue para buscar la reacción pertinente [3 a 23].

Internamente, una vez identificada la ortología, `Gene2Rxn` llama a `RxnDown`, función que descarga la información de la reacción presente en KEGG a partir de la ortología entregada (Ver Anexo 8.2.6).

Habiendo obtenido la información de la reacción por `RxnDown`, la función `Gene2Rxn` escribe un string con todos los genes que se determinó que eran necesarios, en el formato pedido por Cobrapy [25 a 44]. Posteriormente, la función llama a `AddRxn`, quien agrega finalmente la reacción al modelo (ver Anexo 8.2.8). Finalmente, si es que se requiere, la función llama a `WriteByMe` para registrar los cambios realizados en una planilla de cálculo (ver Anexo 8.2.10).

8.2.6 RxnDown

Tabla 15. Resumen de la función RxnDown.

Entradas		
Variable	Tipo	Significado
`ko`	String.	Ortología de KEGG asociada a la reacción.
`ABM`	Lista con un string y dos valores numéricos.	Información de planilla de cálculo: El string es el nombre de la hoja; el primer valor es la primera fila a leer; el segundo valor es el número identificador de la reacción. (Necesario solo en modo automático).
En modo interactivo, requiere entradas numéricas de tipo `input`.		
Salidas		
Variable	Tipo	Significado
`tinfo`	Lista con un string y dos valores numéricos.	El string es el nombre de la reacción; el primer valor es la dirección de la reacción (con respecto a KEGG); el segundo valor es la reversibilidad de la reacción.

Esta función tiene dos métodos de funcionamiento: Interactivo y automático. En el primer mecanismo de operación, se descarga la ortología desde KEGG y se selecciona la información relativa a las reacciones propias de tal ortología [2 a 5]. Posteriormente, a menos que se haya encontrado solo una reacción, el programa le muestra el usuario una lista con todas las reacciones, presentando de cada una de ellas el código de KEGG y la estequiometría, además de un número de identificación propio del programa [6 a 15]. Una vez presentadas las reacciones, se le da al usuario la tarea de definir cuál es la reacción que efectivamente se quiere agregar al modelo [19]. Posteriormente, se le pregunta al usuario si es que la reacción es reversible o irreversible, considerando que esta propiedad está sugerida en los mapas de KEGG pero no está almacenada en la información de reacciones del mismo. Se da a elegir entre 0 (irreversible) y 1 (reversible) 8.7.6[22]. En caso de ser irreversible, es posible que la reacción presente en KEGG vaya en sentido contrario al requerido para el modelo, dado que en la base de datos están todas las reacciones almacenadas como reversibles y con un orden reactante-producto fijo, que podría no corresponder a lo que se necesita. En este caso, el programa le pregunta al usuario si es que el orden está correcto, teniendo que responder con un 1 (sí) o un 0 (no) [27]. Realizadas todas las preguntas pertinentes, `RxnDown` le entrega a `Gene2Rxn` la lista con la información pertinente señalada en la Tabla 15.

En la versión automática de esta función, las tres preguntas realizadas al usuario se responden en base a información almacenada previamente en la variable `ABM`, provenientes de una planilla de cálculo, ya sea llenada por el

usuario o por el sistema [20, 23 y 30]. La información contenida en la variable `ABM` es traspasada a `RxnDown` mediante la función `AnswerByMe` (ver Anexo 8.2.7), para luego entregar la misma lista de respuestas a `Gene2Rxn`.

8.2.7 AnswerByMe

Tabla 16. Resumen de la función AnswerByMe.

Entradas		
Variable	Tipo	Significado
`pagina`	String.	Nombre de la hoja de la planilla de cálculo a usar.
`inirow`	Valor numérico.	Número de la primera fila de la hoja a utilizar.
`task`	String.	Letra que indica el tipo de información a responder.
`rxnn`	Valor numérico.	Número de la reacción estudiada.
`varia`	Puede ser:	
	Lista de strings.	Nombres de las reacciones de la misma ortología.
	Valor `None`.	Es vacío porque no se ocupa en la función.
	String.	Detalle estequiométrico de la reacción estudiada.
Salidas		
`i`	Valor numérico.	Puede ser el número de la reacción; un indicador de reversibilidad (1 = reversible, 0 = irreversible); o un indicador de sentido (1 = de izquierda a derecha, -1 = de derecha a izquierda)

La función AnswerByMe es la encargada de realizar de manera automática ciertas respuestas que COBRAPy requiere respondidas para sumar reacciones al modelo, utilizando una planilla de cálculo con los datos requeridos. Las variables `pagina`, `inirow`, y `rxnn` son indicadores que marcan la celda a la cual extraerle información (ver Tabla 16.); mientras que la variable `task` define la tarea a realizar, y lo que la variable `varia` debe ser.

Cuando `task` es `E` [4 a 10], la pregunta a responder es el número de la reacción a utilizar, y `varia` corresponde a la lista de reacciones agrupadas bajo la misma ortología. El código de la reacción debe estar ya registrado en la planilla de cálculo (ver Anexo 8.3), y lo que hace esta función es comparar tal código con aquellos en `varia` para definir la reacción a utilizar en los siguientes pasos. La función entrega una variable numérica que corresponde al índice de la reacción.

Cuando `task` es `H` [11 a 19], la pregunta es la reversibilidad y el parámetro `varia` es un `None`, porque no se ocupa. En este caso, la función lee

directamente un "si" o un "no" de la planilla y lo convierte en un 1 (reversible) o un 0 (irreversible), que se da como salida de la función.

Cuando `task` es `F` [20 a 55], la pregunta es la dirección y `varia` es la ecuación almacenada en KEGG. Mediante operaciones de manejo de strings, la función separa la ecuación y determina la parte "izquierda" (reactantes) de la "derecha" (productos), para luego comparar con la planilla, que debe tener registardos a un reactante y un producto de la reacción. Al comparar ambas partes, la función entrega un 1 si el orden es de izquierda a derecha, o un 0 si la reacción va en sentido inverso. Adicionalmente, esta función incluye una opción de ingresar el sentido manualmente en caso de fallar la operación de determinación.

8.2.8 AddRxn

Tabla 17. Resumen de la función AddRxn.

Entradas		
Variable	Tipo	Significado
`modelo`	Objeto `model` de COBRAPy.	Modelo al cual se le agregará la reacción.
`rxnID`	String.	Código KEGG de la reacción a agregar.
`dire`	Valor numérico.	Dirección a la cual se debe leer la reacción.
`rev`	Valor numérico.	Reversibilidad de la reacción.
`genstr`	String.	Genes que regulan la reacción.
Salidas		
Esta función no genera salidas.		

La función `AddRxn` es, básicamente, la encargada de agregar las reacciones al modelo. Utilizando el identificador (`rxnID`, ver Tabla 17), la función descarga la información disponible de la reacción desde KEGG, y la subdivide para su posterior procesamiento [4 y 5].Luego , genera un elemento `Reaction` de Cobrapy usando el `rxnID` como nombre identificador, y le agrega el nombre descriptivo (descargado desde KEGG, o agregado por el usuario en caso de que éste falte). Usando como fuente la variable `rev`, la función genera los límites de la reacción para que ésta sea reversible o irreversible [14 a 18].

A partir de la información estequiométrica de la reacción, que viene como un único string, se obtienen los metabolitos involucrados, con sus respectivos coeficientes [19 a 62]. Posteriormente, se llama a la función `AddMet` (Anexo 8.2.9) para agregar los metabolitos al modelo, o bien llamar a los compuestos que ya están en éste [38 y 58]. Luego, se generan los coeficientes estequiométricos de cada metabolito, colocando números negativos a los reactantes, y números positivos a los productos, con la

posibilidad de invertir los valores con mediante el uso de la variable `dire` [39 y 59]. Una vez listos los metabolitos y sus coeficientes, se agregan a la reacción, dando un aviso al usuario. Para finalizar, se le agrega a la reacción los genes que la regulan [63 a 65], los cuales vienen en la variable `genstr`.

Cabe mencionar que esta función da cabida a ingresar reacciones como gap, para lo cual se ingresa como `genstr` un string de largo 1, como por ejemplo un espacio. Al hacer tal ingreso, la función dejará la información de los genes en blanco.

Después de ingresar (o no) los genes, la reacción es finalmente agregada al modelo, dándole un aviso al usuario. En caso de estar la reacción ya agregada al modelo, la función no opera, y le entrega al usuario un aviso de la situación.

8.2.9 AddMet

Tabla 18. Resumen de la función AddMet

Entradas		
Variable	Tipo	Significado
`mode`	Objeto `model` de COBRAPy.	Modelo al cual se le agregará el metabolito.
`metID`	String.	Código KEGG del metabolito.
Salidas		
Variable	Tipo	Significado
`mmet`	Objeto `metabolite` de COBRAPy.	Metabolito sobre el cual se está trabajando.

La función AddMet es la encargada de agregar metabolitos al modelo para su uso en las reacciones, o bien simplemente se encarga de entregarle a la reacción el metabolito ya almacenado.

Cuando el metabolito no se encuentra en el modelo, se utiliza una clase KEGG de Bioservices para descargar y desglosar la información del metabolito [5 y 6], tomando particularmente el nombre y la fórmula del metabolito (la función puede pedir al usuario que ingrese la fórmula manualmente en caso de no estar disponible ésta en KEGG), en base al índice entregado a la función (ver Tabla 18). Posteriomente, la función genera el nuevo metabolito bajo la variable `mmet`, utilizando la información desglosada [13].

En caso de ya estar el metabolito presente en el modelo, la función le avisa al usuario de esta situación, y devuelve el metabolito requerido sin generarlo de nuevo [16 y 17].

8.2.10 WriteByMe

Tabla 19. Resumen de la función WriteByMe.

Entradas		
Variable	Tipo	Significado
string	String.	Genes a anotar en la planilla de cálculo.
info	Lista de un string y 2 valores numéricos.	El string es el nombre de la hoja de la planilla; el primer valor es la primera fila a leer; el segundo valor es el número identificador de la reacción cuyos genes se anotarán.
Salidas		
Esta función no genera salidas. Sin embargo, escribe un fragmento de una planilla de cálculo.		

Esta función escribe en la planilla de cálculo los genes utilizados en las reacciones correspondientes. Como se ve en el Anexo 8.7.10, la función toma la variable `info` descrita en la Tabla 19 para posicionarse dentro de la hoja. Una vez localizado el espacio, éste se rellena con el string contenedor de los genes.

8.3 Descripción de generación de planilla de información

Tabla 20. Resumen de la función xlsMaker

Entradas		
Variable	Tipo	Significado
`xml`	String.	Nombre del archivo KGML (en formato .xml) con la información a agregar.
`xls`	String.	Nombre de la planilla de cálculo a editar.
`hoja`	String.	Nombre de la hoja a editar de la planilla de cálculo.
Salidas		
Esta función no genera salidas. Sin embargo, edita una planilla de cálculo.		

Tabla 21. Resumen de la función GetDef.

Entradas		
Variable	Tipo	Significado
`code`	String.	Código KEGG de la reacción a analizar.
Salidas		
Variable	Tipo	Significado

defi	String.	Definición de la estequiometría de la reacción.

Tabla 22. Resumen de la función WriteEverything.

Entradas		
Variable	Tipo	Significado
hoja	String.	Nombre de la hoja a editar de la planilla de cálculo.
index	Valor numérico.	Indicador de la primera fila de la hoja a editar.
rxn	String.	Código KEGG de la reacción a escribir.
s	String.	Código KEGG de algún sustrato de la reacción.
p	String.	Código KEGG de algún producto de la reacción.
rev	Valor numérico.	Indicador de reversibilidad de la reacción.
defi	Valor numérico.	Dirección de la reacción en el modelo.
genstr	String.	Genes del organismo base involucrados en la reacción.
Salidas		
Esta función no genera salidas. Sin embargo, edita una planilla de cálculo.		

La función xlsMaker, después de tomar los argumentos especificados en la Tabla 20, abre los archivos pertinentes en base a los strings entregados, y localiza la hoja correspondiente en la planilla. Idealmente, la hoja vacía ya debería existir en el archivo.

Mediante el KGML_parser que viene incluido en Biopython (importado como Kp para simplificar), se procesa el archivo KGML, obteniendo todas las reacciones representadas en el mapa que están presentes en el organismo seleccionado. Para cada reacción, se obtiene el parámetro entry.name, que es un string que alberga a los nombres de los genes que se encargan de la correspondiente reacción. Mediante un manejo básico de strings, se obtiene la parte numérica de cada nombre de gen y se almacena en un único string llamado genstr. Posteriormente, se almacena el código de la reacción en KEGG bajo la variable name0, y se guardan los códigos del primer reactante y del primer producto que aparecen en la reacción bajo las variables sub y prod, respectivamente. Cabe mencionar que la razón para agregar solo uno de todos los productos y reactantes, es que con ellos solo se pretende otorgarle la dirección a la reacción mas que definirla completamente. Se almacena a partir del carácter de índice 4 puesto que en el KGML los productos y reactantes están referidos bajo la forma "cpd:CXXXXX", por lo cual se le elimina el "cpd:" por motivos prácticos. Finalmente, se lee la reversibilidad de la reacción, almacenada bajo el campo type de la reacción,

y se almacena una variable igual a 0 si es irreversible, y 1 en el caso contrario, bajo la variable `rev`.

Con toda la información necesaria almacenada, se chequea el largo de `name0`. Si es de largo 9, quiere decir que hay solo una reacción involucrada, bajo el formato "rn:RXXXXX". A `name0` se le corta la primera parte y se manda el resto, bajo la variable `name`, a la función `WriteEverything` junto con los parámetros ya mencionados, más una variable `index` que se actualiza en cada iteración del proceso y que marca la fila que ocupará la información en la planilla; adicionalmente, se usa la función `GetDef` para obtener la definición estequiométrica de la reacción, la cual se guarda en la variable `defi`. Si `name0` no mide 9 caracteres, significa que hay dos o más reacciones representadas en la misma vía (Ejemplo: Una reacción que se puede llevar a cabo ocupando indistintamente NAD o NADP aparecerá dos veces en el sistema; una vez usando NAD y otra usando NADP), para lo cual se divide `name0` usando la función de Python `split()` y usando como separador el espacio, de modo tal de obtener un string distinto para cada código de reacción presente. Luego, mediante un ciclo `for`, se realiza el proceso descrito para el caso de largo 9, para cada reacción presente en `name0`, a modo de obtener todas las reacciones presentes en la vía.

La función `GetDef`, tal como sugiere la Tabla 21, toma el código de la reacción, y mediante un módulo KEGG de bioservices, descarga la información de la reacción, la secciona, y entrega el desarrollo estequiométrico de la reacción bajo la variable `defi`. Por su parte, la función `WriteEverything` ingresa todos los parámetros anotados en la Tabla 22 a la hoja de cálculo, colocando desde la columna B hasta la H los siguientes parámetros en orden: Número de la reacción en la hoja (`index`); genes de *S. avermitilis* asociados a la reacción (`genstr`), definición estequiométrica de la reacción (`defi`); código de la reacción (`rxn`); el código de uno de los reactantes (`s`); el código de uno de los productos (`p`); y un "sí" o un "no" dependiendo del parámetro de reversibilidad (`rev`). Una vez agregados todos los parámetros, se guarda la planilla, tomando la hoja el aspecto que se aprecia en la Figura 5.

B	C	D	E	F	G	H
N° de reacción	Genes	Reacción	Código reacción	Inicio	Fin	Rev?
1	[2130, 2312],	ATP + Adenylyl sulfate <=> ADP + 3'-Phosphoadenylyl sulfate	R00509	C00224	C00053	no
2	[2131, 2313, 2314],	ATP + Sulfate <=> Diphosphate + Adenylyl sulfate	R00529	C00059	C00224	si
3	[1435],	O-Acetyl-L-serine + Hydrogen sulfide <=> L-Cysteine + Acetate	R00897	C00979	C00033	no
4	[2129],	Thioredoxin + 3'-Phosphoadenylyl sulfate <=> Thioredoxin disulfide	R02021	C00053	C00094	no
5	[3305],	O-Succinyl-L-homoserine + Hydrogen sulfide <=> L-Homocysteine + S	R01288	C00283	C00042	no
6	[2127],	Hydrogen sulfide + 6 Oxidized ferredoxin + 3 H2O <=> Sulfite + 6 Red	R00859	C00283	C00094	si

Figura 5. Visualización final de la planilla de cálculo posterior a su llenado automático en base al KGML. Particularmente, este es el resultado de sma00920: Metabolismo de azufre de S. avermitilis.

Los códigos de las tres reacciones utilizadas están disponibles en los Anexos 8.7.11, 8.7.12, y 8.7.13

8.4 Detalle de reacciones de biomasa

8.4.1 Biomasa

En este trabajo, se utilizaron las proporciones correspondientes a *Streptomyces clavuligerus*, las cuales son similares a las de *S. avermitilis* (Toro et al., 2018). La composición de la biomasa del HST28 fue asumida como la siguiente:

Tabla 23. Composición de la biomasa de HST28 con respecto al peso seco de ésta.

Componente	Proporción [g g^{-1}DW]
Protoplasto:	
Proteínas	0,474
DNA	0,033
RNA	0,057
Lípidos:	
Fosfolípidos	0,056
TAGs	0,084
Moléculas pequeñas	0,03
Pared celular:	
Peptidoglicano	0,095
Carbohidratos	0,087
Ácido teicoico	0,065

Cabe mencionar que, según la misma bibliografía, se requieren 47 moléculas de ATP para generar 1 gramo de biomasa, lo cual debe ser agregado a la estequiometría de la reacción, quedando ésta del siguiente modo:

47 H2O + 47 ATP + 0,474 Proteínas + 0,033 DNA + 0,057 RNA + 0,056 Fosfolípidos + 0,084 TAGs + 0,03 Moléculas pequeñas + 0,095 Peptidoglicano + 0,087 Carbohidratos + 0,065 Ácido teicoico → 1 Biomasa + 47 ADP + 47 PI

8.4.2 Proteínas

Se asumió una composición de proteínas igual a la de *S. avermitilis*, que del mismo modo se ha asumido igual a la de *S. clavuligerus* (Toro et al., 2018). Según tal fuente, se requiere el consumo de 40 ATP para generar un gramo

de proteína. En base a lo previamente mencionado, la composición de las proteínas de HST28 es la siguiente:

Tabla 24. Composición global de las proteínas del HST28.

Aminoácido	Porcentaje c/r al total	Peso molecular [g mol^{-1}]	mmol por gramo de proteína
L-Alanina	8,92%	71,09	1,255
L-Arginina	11,67%	156,2	0,747
L-Asparagina	1,93%	114,12	0,169
L-Aspartato	6,39%	115,1	0,555
L-Cisteína	0,78%	103,16	0,076
L-Fenilalanina	3,75%	147,19	0,255
Glicina	5,06%	57,07	0,886
L-Glutamato	3,41%	128,15	0,266
L-Glutamina	6,77%	129,13	0,524
L-Hisitidina	3,02%	137,16	0,22
L-Isoleucina	3,32%	113,18	0,293
L-Leucina	10,80%	113,18	0,954
L-Lisina	2,76%	128,19	0,215
L-Metionina	2,22%	131,21	0,169
L-Prolina	5,54%	97,13	0,57
L-Serina	4,28%	87,09	0,491
L-Tirosina	3,23%	163,19	0,198
L-Treonina	5,90%	101,12	0,583
L-Triptófano	2,68%	186,23	0,144
L-Valina	7,72%	99,15	0,779

Cabe mencionar que en la Tabla 24 se le sustrajo al peso molecular de los aminoácidos el equivalente al peso de una molécula de agua, para representar la pérdida de H2O al generar los enlaces peptídicos entre los aminoácidos al momento de conformar las proteínas. La reacción de producción de este componente es la siguiente:

1,255 Ala + 0,747 Arg + 0,169 Asn + 0,555 Asp + 0,076 Cys + 0,524 Gln + 0,266 Glu + 0,886 Gly + 0,22 His + 0,293 Ile + 0,954 Leu + 0,215 Lys + 0,169 Met + 0,255 Phe + 0,57 Pro + 0,491 Ser + 0,583 Thr + 0,144 Trp + 0,198 Tyr + 0,779 Val + 40 ATP + 40 H2O →1 Proteína + 40 ADP + 40 PI

8.4.3 DNA

Las proporciones de los nucleótidos en el DNA fueron tomadas a partir de estudios para *S. avermitilis* (Toro et al., 2018). Se consideró un gasto de ATP de 4,4 mmol por gramo de DNA sintetizado (Ingraham, 1983). Con ello, las cantidades son las siguientes:

Tabla 25.Composición global del DNA del HST28.

Nucleótido	mol/mol DNA (%)	PM [g/mol]	mmol/g DNA
dAMP	14,7	313,2	0,474
dCMP	35,4	289,2	1,144
dTMP	14,7	304,2	0,474
dGMP	35,4	329,2	1,144

Quedando la reacción de la siguiente manera:

0,474 dATP + 1,144 dCTP + 0,474 dTTP + 1,144 dGTP + 4,4 ATP + 4,4 H2O →1 DNA + 4,4 ADP + 4,4 PI + 3,236 PPI

8.4.4 RNA

Se asumió un 4% de mRNA, un 81% de rRNA, y un 15% de tRNA en el total de RNA presente en el HST28, acorde con (Toro et al., 2018). Además, se consideró un gasto de ATP de 1,268 mmol por gramo de RNA sintetizado, acorde con la misma fuente. Las cantidades son las siguientes:

Tabla 26. Composición global del RNA del HST28.

Nucleó-tido	mol/mol RNA (%)						PM [g/mol]	mmol/g RNA
	mRNA (4%)	5S rRNA (2%)	16S rRNA (27%)	23S rRNA (52%)	tRNA (15%)	RNA total		
AMP	14,70	19,00	18,20	20,10	19,80	19,30	329,2	0,602
GMP	35,40	24,60	25,10	22,90	29,60	25,03	345,2	0,782
CMP	14,70	32,50	34,10	34,00	34,60	33,31	305,2	1,068
UMP	35,40	23,80	22,60	23,00	16,10	22,37	306,2	0,671

En base a lo ya expuesto, y considerando que el ATP se ocupa tanto como proveedor de energía como precursor del AMP, la reacción queda del siguiente modo:

1,87 ATP + 0,782 GTP + 1,068 CTP + 0,671 UTP + 1,268 H2O → 1 RNA + 1,268 ADP + 1,268 PI + 3,123 PPI

8.4.5 Fosfolípidos

En el modelo, se consideraron tres tipos de fosfolípido: Fosfatidiletanolamina (PE), fosfatidilglicerol (PG) y cardiolipina (CL), cuyas proporciones relativas, dadas por (Borodina et al., 2005), están en la Tabla 27. Cabe mencionar que estas tres moléculas son genéricas, vale decir, no tienen una composición molecular establecida, si no que representan a una serie de moléculas que comparten una estructura similar. Las formas base

de las tres especies están en la Figura 6, en donde las letras R representan los radicales que pueden variar de una molécula a otra. Particularmente para este caso, tales radicales consisten exclusivamente de cadenas de ácidos grasos.

Tabla 27. Composición de los fosfolípidos del HST28.

Metabolito	Proporción (%)	mmol/g
PE	75	1,089
PG	18	0,250
CL	7	0,052

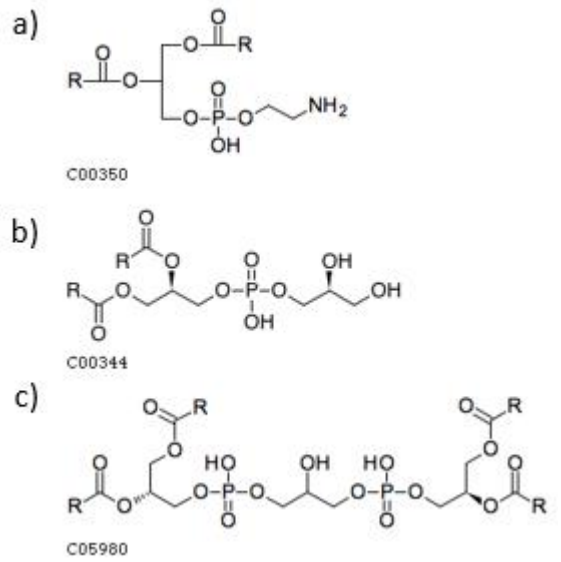

Figura 6. Estructuras de los fosfolípidos componentes del HST28. a) Fosfatidiletanolamina. b) Fosfatidilglicerol. c) Cardiolipina. (Fuente: KEGG).

Las proporciones de ácidos grasos presentes en estas moléculas fueron asumidas como idénticas a aquellas reportadas por (Borodina et al., 2005). Tales cantidades están representadas en la Tabla 28. Cabe mencionar que el proceso de conformación de los fosfolípidos acorde a las proporciones entregadas no es directo, si no que se requieren diversos pasos para obtener el resultado final. A partir del glicerol-3P, se forma el 1,2-diacil-glicerol-3P mediante la adición de los ácidos grasos en las proporciones definidas en dos pasos, siendo las reacciones:

1 G3P + 0,094 C14-CoA + 0,294 C15-CoA + 0,262 C16-CoA + 0,293 C17-CoA + 0,057 C18-CoA → 1 1-acil-G3P + 1 CoA

1 1-acil-G3P + 0,094 C14-CoA + 0,294 C15-CoA + 0,262 C16-CoA + 0,293 C17-CoA + 0,057 C18-CoA → 1 1,2-diacil-G3P + 1 CoA

Tabla 28. Proporción de ácidos grasos (AG) en los fosfolípidos del HST28, según el largo de su cadena de carbono.

AG	g/g total AG	PM [g/mol]	mmol/g total AG	mol/mol total AG
C14	0,084	227	0,37	0,094
C15	0,279	241	1,16	0,294
C16	0,263	255	1,03	0,262
C17	0,311	269	1,16	0,293
C18	0,063	281	0,22	0,057

Con el 1,2-diacil-glicerol-3P conformado acorde a las proporciones dadas, se llevan a cabo las reacciones que se ven en la Figura 7, las cuales se resumen de la siguiente manera:

General:

1 1,2-diacil-G3P + 1 CTP → 1 CDP-diacilglicerol + 1 PPI

Hacia PE:

1 CDP-diacilglicerol + 1 L-Serina → 1 PS + 1 CMP

1 PS → 1 PE + 1 CO2

Hacia CL:

1 CDP-diacilglicerol + 1 G3P → 1 PGP + 1 CMP

1 PGP + 1 H2O → 1 PG + 1 PI

1 CDP-diacilglicerol + 1 PG → 1 CL + 1 CMP

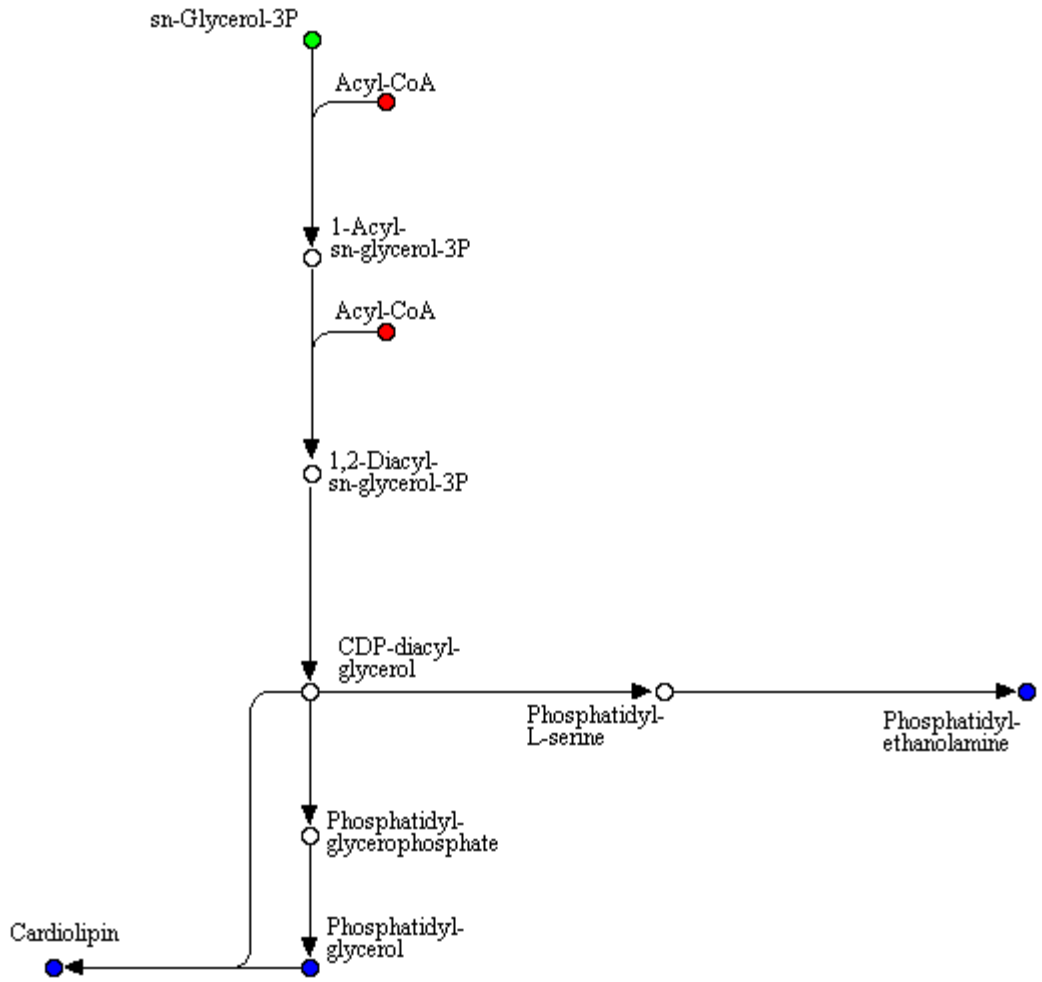

Figura 7. Fragmento simplificado de la vía de los fosfolípidos en KEGG. En verde está el metabolito base del proceso. En rojo están las adiciones de ácidos grasos. En azul están los metabolitos finales de la vía.

Con los tres metabolitos conformados, la reacción de producción de fosfolípidos es la siguiente:

1,089 PE + 0,25 PG + 0,052 CL → 1 Fosfolípido

8.4.6 TAGs

Las proporciones utilizadas de ácidos grasos en los triacilgliceroles (TAGs) fueron las mismas utilizadas en el modelo de *S. avermitilis* (Toro et al., 2018), las cuales están en la Tabla 29.

Tabla 29. Composición de los triacilgliceroles del HST28.

Componente	mol/mol TAG	mmol/g TAG
G3P	1	1,244
C14	0,040	0,050
C15	1,349	1,677
C16	0,338	0,421
C17	1,262	1,570

C18	0,011	0,014

En base a los datos presentes, la reacción es la siguiente:

1,244 G3P + 0,05 C14-ACP + 1,677 C15-ACP + 0,421 C16-ACP + 1,57 C17-ACP + 0,014 C18-ACP -> 1 TAG + 1,244 PI + 3,732 ACP

8.4.7 Moléculas pequeñas

Las moléculas pequeñas son ocho en total: ACP, CoA, FAD, FMN, menaquinona (MK), NAD, NADP y tetrahidrofolato (THF). Se asumió que todas las especies están representadas igualmente en relación a peso, vale decir, 0,125 gramos de cada molécula por gramo total de este ítem. Con tal dato a considerar, la cantidad de cada especie en milimoles por gramo son las siguientes:

Tabla 30. Composición de las moléculas pequeñas del HST28.

Molécula	PM [g/mol]	Cantidad [mmol/g]
ACP	10689	0,012
CoA	767,534	0,163
FAD	785,557	0,159
FMN	456,348	0,274
MK	853,365	0,146
NAD	664,438	0,188
NADP	744,418	0,168
THF	445,434	0,281

La reacción es juntar directamente todos los compuestos, quedando de la siguiente manera:

0,012 ACP + 0,163 CoA + 0,159 FAD + 0,274 FMN + 0,146 MK + 0,188 NAD + 0,168 NADP + 0,281 THF → 1 Moléculas pequeñas

8.4.8 Peptidoglicano

Los componentes del peptidoglicano presente en HST28 fueron asumidos como idénticos a aquellos regsitrados para *S. avermitilis* y *S. clavuligerus* (Toro et al., 2018), que a su vez son análogos al de *S. coelicolor* (Borodina et al., 2005). Los valores fueron generados en base a un promedio entre tres especies de *Streptomyces*, particularmente *S. chrysomallus*, *S. antibioticus* y *S. roseoflavus*. Los componentes son los siguientes:

Tabla 31. Composición del peptidoglicano del HST28.

Componente	Razón Molar	PM [g/mol]	Cantidad [mmol/g]
N-Acetilmuramato	0,9	275	1,007
N-Acetilglucosamina	1,1	203	1,197
Alanina	1,7	71	1,900
Ácido diaminopimelínico	1,0	154	1,140
D-Glutamato	0,9	129	1,014
Glicina	0,9	39	0,973

Para determinar la reacción, hay que considerar que tanto el N-Acetilmuramato como la N-Acetilglucosamina reaccionan en una forma activada con UDP (UDP-NAM y UDP-NAG, respectivamente). Se asume además que la mitad de la alanina presente en el peptidoglicano está como D-Alanina, la cual viene en forma de D-Alanil-D-Alanina, liberando una molecula de D-Alanina al medio extracelular en el proceso de enlace de las cadenas de NAM y NAG. Con un consumo de ATP asumido de 5,026 mmol/gramo de peptidoglicano producido, la reacción es la siguiente:

1,007 UDP-NAM + 1,197 UDP-NAG + 1,14 Ácido diaminoímelínico + 0,95 Ala + 1,9 D-Ala-D-Ala + 1,014 D-Glu + 0,973 Gly + 5,026 ATP → 1 Peptidoglicano + 1,007 UMP + 1,197 UDP + 0,95 D-Ala(ext) + 5,026 ADP + 5,026 PI

8.4.9 Carbohidratos

Se asumió que los carbohidratos presentes en la pared celular eran idénticos en proporción a los de *S. clavuligerus* (Toro et al., 2018), correspondiendo a galactosa y N-acetilglucosamina, estando éstos en las proporciones señaladas en la Tabla 32.

Tabla 32. Composición de carbohidratos en la pared celular de HST28.

Componente	Razón molar	PM [g/mol]	mmol/g carbohidratos
N-Acetilglucosamina	1	203	1,897
Galactosa	2	162	3,794

Ambos componentes se presentan en su forma activada por UDP (UDP-NAG y UDP-Gal, respectivamente). Con ello, la reacción queda de la siguiente manera:

1,897 UDP-NAG + 3,794 UDP-Gal → 1 Carbohidratos + 5,691 UDP

8.4.10 Ácido teicoico

La información para configurar la producción de ácido teicoico se obtuvo de (Borodina et al., 2005), en donde se asume un largo promedio de las cadenas del ácido igual a 12 unidades de glicerofosfato. Adicionalmente, se asume que un 25% de estas cadenas tendría un residuo de N-acetilglucosamina, y un 25% tendría un residuo de L-Lisina, acorde con promedios registrados en diversos *Streptomyces*. Así, la composición es la siguiente:

Tabla 33. Composición del ácido teicoico de HST28.

Componentes	Razón molar	PM [g/mol]	mmol/g ácido
Cadena Polygp	1	1849	0,518
L-Lisina	0,25	128	0,129
N-Acetilglucosamina	0,25	203	0,129

Cabe mencionar que la generación del ácido teicoico toma dos pasos en el modelo. El primero de ellos es la formación de la cadena de poliglicerol (Polygp), la cual se hace a partir de doce moléculas de CDP-Glicerol según:

12 CDP-Glicerol → 1 Polygp + 12 CMP

Con ello, más un consumo de ATP definido como 0,129 mmol por gramo de ácido, la reacción se expresa de la siguiente manera:

0,518 Polygp + 0,129 UDP-NAG + 0,129 Lys + 0,129 ATP → 1 Ácido teicoico + 0,129 UDP + 0,129 ADP + 0,129 PI

8.5 Método de adición de reacciones de intercambio y transporte (EXmaker)

El código utilizado para esta operación se encuentra en el Anexo 8.7.11. Las referencias en corchetes ("[]") corresponden a la línea de código de la cual se está hablando.

Tabla 34. Resumen de la función EXmaker.

Entradas		
Variable	Tipo	Significado
`modelo`	Objeto `model` de COBRAPy.	Modelo al cual se le agregarán las reacciones.

`codigo`	String.	Identificador KEGG del metabolito en torno al cual se harán las reacciones.
`tipo`	Valor numérico.	Indicador del tipo de reacción de transporte.
Salidas		
Esta función no genera salidas.		

Lo primero que hace esta función es tomar el `codigo` (Ver Tabla 34) y buscar el metabolito dentro del modelo que comparte el mismo identificador [2]. Posteriormente, la función clona el metabolito, pero con la diferencia de estar en el compartimento `e` (extracelular) en vez de `c` (citoplasma), lo cual es también representado en el nombre del nuevo metabolito. En caso de ya existir tal metabolito extracelular, la función tan solo se limita a llamarlo [3 a 7].

Una vez generado el metabolito extracelular, la función genera la reacción de transporte. La naturaleza de ésta dependerá del valor de `tipo`. Si `tipo` es 0, la reacción corresponde a una difusión simple o facilitada. En este caso, la función se limita a generar una reacción reversible con parámetros estándar, en dónde los metabolitos son solo dos: La molécula extracelular que se consume, y la molecula original que se genera (se asume que esta es una entrada a la célula) [8 a 20].

Si `tipo` es igual a 1, la reacción corresponde a un transporte mediado por ATP. Con pasos idénticos a los seguidos para el caso de la difusión, la función genera una reacción irreversible en la cual se consume el metabolito exterior junto a agua y ATP, y se genera el metabolito interior junto con ADP y ortofosfato [21 a 37].

Cuando `tipo` es igual a 2, la reacción corresponde a un simporte de protones En este caso, la función le preguntará al usuario los parámetros de reversibilidad [47], número de protones requeridos para hacer funcionar el simporte [50], y sentido de la reacción (ingreso o salida) [51]. Con tal información, la función genera la reacción y luego agrega los metabolitos, considerando protones externos que entran (o salen) junto con el metabolito a transportar. Así, hay una molécula y protones a ambos lados de la reacción, y por ende, de la membrana celular [52 a 55].

Finalmente, cuando `tipo` es igual a 3, la reacción es un simporte con sodio. En este caso, la función genera una reacción irreversible que consume al metabolito exterior en conjunto con sodio exterior, para generar el metabolito intracelular junto con sodio intracelular [56 a 70].

Cabe mencionar que, para los cuatro tipos de reacción, la función da espacio a que hayan más de un transporte de la misma clase para el mismo metabolito. Para ello, permite ingresar un identificador extra al nombre de

la reacción a agregar, para poder diferenciarla de aquella ya agregada [10 a 13, 23 a 26, 40 a 43, 58 a 61].

Adicionalmente, la función genera, en caso se que no esté, la reacción de intercambio para el metabolito señalado. Para ello, simplemente genera una reacción reversible que consta únicamente del consumo del metabolito exterior [72 a 78]. Esta reacción se almacena con el sufijo "EX_" seguido del identificador KEGG del metabolito.

Actualmente, el código no genera reacciones de antiporte ni mediados por fosfotransferasa. Tales reacciones han de agregarse manualmente.

8.6 Método de rastreo de errores (errTracker)

El código utilizado para esta operación se encuentra en el Anexo 8.7.15. Las referencias en corchetes ("[]") corresponden a la línea de código de la cual se está hablando.

Tabla 35. Resumen de la función errTracker.

Entradas		
Variable	Tipo	Significado
`modelo`	Objeto `model` de COBRApy.	Modelo al cual se le rastrean los errores.
`rxn`	String.	Nombre de reacción del modelo que se somete al rastreo.
`xls`	String.	Nombre de planilla de cálculo en donde se registran los resultados del rastreo.
Salidas		
Esta función no genera salidas. Sin embargo, actualiza una hoja de una planilla de cálculo.		

En primer lugar, es necesario definir la estructura que debe tener la planilla de cálculo de la cual se hace referencia en la Tabla 35. Ésta debe tener dos hojas, llamadas "Raw", que debe estar vacía; y "Base", que debe tener una serie de números ordenados en matrices de potencias de 2, como se ve en la Figura 8. En el ejemplo, se comienza por el 0 en la esquina superior izquierda; luego, a la derecha se le coloca el mismo número, pero sumado en 2^0, lo que da 1; posteriormente en los cuadros que están debajo de los dos ya citados, se colocan los números que ya se tienen, pero sumados en 2^1, lo que da 2 debajo de 0 y 3 debajo de 1; teniendo ya un espacio de 2x2 cuadros, se toma el espacio de 2x2 contiguo a la derecha y, de manera correlativa, se colocan los mismos números, pero sumados en 2^2, llegando a 7. Este proceso se itera hasta 2^5 para llegar al ejemplo de la Figura 8. En la Figura 9 Se demuestra de manera más clara el mismo proceso para una tabla más pequeña. En el caso de este trabajo, se utilizaron 10 iteraciones,

llegando a 1024 valores. Para ocupar la planilla en la función, es necesario que los números estén como valor simple y no como fórmula.

0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

Figura 8. Ejemplo de distribución de números en la hoja "Base".

0	0 + 1 = 1	0 + 4 = 4	1 + 4 = 5
0 + 2 = 2	1 + 2 = 3	2 + 4 = 6	3 + 4 = 7
0 + 8 = 8	1 + 8 = 9	4 + 8 = 12	5 + 8 = 13
2 + 8 = 10	3 + 8 = 11	6 + 8 = 14	7 + 8 = 15

Figura 9. Detalle de la numeración de la hoja "Base". En rojo está la zona en la que se agrega 2^0. En amarillo, se agrega 2^1. En verde, se agrega 2^2.Y en celeste, se agrega 2^3.

La función `errTracker` analiza las entradas y salidas de una reacción que no puede ser optimizada, para descubrir la falla presente en el modelo que genera esta incapacidad. Lo primero que hace la función es identificar la reacción a examinar [2] y leer la planilla de cálculo y sus hojas [4 a 6]. Luego, se leen los metabolitos de la reacción y se agregan a una lista aquellos que no tienen una reacción de intercambio asociada (se asume que el metabolito tiene un nombre de estilo CXXXXX_c y que las reacciones de intercambio se nombran por EX_CXXXXX) [7 a 12]. Posteriormente, la función le muestra al usuario el orden en el cual los metabolitos fueron agregados a la lista, puesto que el modelo no siempre entrega los metabolitos de las reacciones en el mismo orden cuando se les requiere [14].

Posteriormente, la función lee las celdas de la hoja "Base", y transforma el valor presente en ellas a número binario, estandarizando con ceros a la izquierda a los números más cortos [16 a 23]. Cabe mencionar que este procedimiento se hará para las celdas cuyo número binario tenga menor o igual cantidad de dígitos que el largo de la lista con los metabolitos [20].

Teniendo el número binario respectivo de una celda, la función crea un contexto (copia temporal) del modelo mediante el comando `with` [24], y para cada metabolito, toma el índice de éste en la lista, e identifica el dígito del numero binario que comparte esa posición. Si el valor en tal espacio es 1, la función genera una reacción de intercambio y transporte para tal metabolito utilizando `EXmaker` (ver Anexo 8.5), contando de derecha a izquierda [24 a 30]. Por ejemplo, en una reacción con cuatro metabolitos, si el número binario es 0101, se hará el transporte e intercambio para los compuestos con índices 1 y 3.

Manteniéndose aún en referencia a la misma celda, dentro del contexto dado por `with`, la función define la reacción problema como objetivo [31], optimiza el modelo [32], y almacena el valor resultante en la celda de "Raw" equivalente a la celda de "Base" actual [36]. Este proceso se realiza para todas las celdas cuyo número es menor a 2^n, con n igual al número de metabolitos seleccionados anteriormente. De este modo, se obtienen todas las combinaciones posibles entre metabolitos con salidas y metabolitos sin salidas. Finalmente, la función salva la planilla de cálculo y la operación termina [37].

Habiendo terminado la parte automatizada del proceso, el usuario debe hacer la segunda parte de este protocolo. En primer lugar, utilizando el orden de metabolitos que el programa entrega, el usuario debe hacer la tabla final del proceso, en la cual se identifica en los bordes qué metabolitos tienen salida en cada celda. Además, por razones visuales, se pueden colocar bordes distintos a cada subconjunto de la tabla para facilitar el distinguir cada metabolito en análisis. Con la tabla arreglada, el usuario debe pegar los valores desde "Raw" a esta, y opcionalmente se pueden colorear las celdas para distinguir aquellas en las que hubo un número favorable, de aquellas en las que el modelo no se optimizó (valor 0).

		C00003_c	C06032_c	
	0	0	1000	1000
C00004_c	0	0	1000	1000
C00109_c	0	0	1000	1000
	0	0	1000	1000
		C01050_c	C04631_c	
	0	375	0	1000
C00003_c	0	375	0	1000
C00004_c	0	375	0	1000
	0	375	0	1000

Figura 10. Ejemplos de tablas de rastreo de errores.

El paso siguiente es interpretar las tablas. En la esquina superior izquierda (Ver Figura 10), se encuentra la situación original, sin salidas adicionales para ningún metabolito. Esta celda suele tener valor 0, puesto que indica que la optimización sobre esa reacción no funciona. Por otra parte, en la esquina inferior derecha, está la situación en la cual todos los metabolitos tienen salidas (o entradas) al medio externo. Este valor siempre es mayor a cero, puesto que no depende del modelo en cuanto todo su funcionamiento está exteriorizado. El resto de los valores es el fruto de las combinaciones posibles entre metabolitos con y sin salida. Para identificar esto, se comienza por los títulos que abarcan más espacio: Todas las celdas que están debajo de un título que abarca dos espacios tienen una salida para el metabolito del título. Luego hay que ver las subdivisiones: Todas las

celdas que están debajo de un título de un espacio tienen una salida para tal metabolito, pero además, la columna equivalente debajo del título doble también tiene salida para este metabolito. La misma lógica se aplica para la filas. Por ejemplo, en la primera tabla de la Figura 10, la celda en la cuarta columna y tercera fila tiene salidas para C06032_c, C00109_c, y C00003_c.

El concepto detrás de este método es que, si una reacción que no está funcionando apropiadamente se "arregla" al agregarle una salida para cierto metabolito, es porque es ese el metabolito clave que evita que la optimización funcione. En los ejemplos de la Figura 10, es un solo metabolito el que genera el problema, siendo C06032_c en el primero y C01050_c en el segundo. La idea es posteriormente intentar deducir por qué tal metabolito entorpece el funcionamiento del modelo, y para ello se pueden estudiar las otras vías en la que aparecen los metabolitos problemáticos, usando este mismo sistema.

8.7 Códigos utilizados

8.7.1 RxnInter

```
def RxnInter(ggroups, name, modelo, excel, exdata): #Funciona solo
 para Streptomyces avermitilis
    sheet = excel[exdata[0]]
    toABM = [sheet, exdata[1], 0]
    fgttn = []
    pgenes = []
    for ggroup in ggroups:
        for gene in ggroup:
            if gene not in pgenes:
                pgenes.append(gene)
    genes = []
    for pgen in pgenes:
        gen = 'SAVERM_' + str(pgen)
        genes.append(gen)
    FastaMaker(genes, name)
    Blaster(name)
    handle = open(name + '.xml')
    result = NCBIXML.parse(handle)
    result = list(result)
    e_tres = 0.001
    i_tres = 0.45
    i = 1
    for ggroup in ggroups:
        hsts = []
        toABM[2] = i
        print '***REACTION ',str(i),' ***'
        for gene in ggroup:
```

```
            print '***GEN ',str(gene),' ***'
            res = HSTRetrieval(gene, result)
            hsts.append(res)
        if None in hsts:
            goon = input('Hay un gen desconocido, seguir? y=1, n=0
')
            if goon == 0:
                fgttn.append(ggroup)
                print 'Reaccion guardada en pendientes'
            else:
                print 'Buscando reaccion ', str(i)
                Gene2Rxn(ggroup, hsts, i, modelo, toABM)
                excel.save('notas.xlsx')
        else:
            print 'Buscando reaccion ',str(i)
            Gene2Rxn(ggroup, hsts, i, modelo, toABM)
            excel.save('notas.xlsx')
        i = i + 1
    handle.close()
    print '***OPERACION TERMINADA***'
```

8.7.2 FastaMaker

```
def FastaMaker(genes, name):
    k = KEGG()
    tofasta = []
    for gen in genes:
        kgen = k.get('sma:'+gen, 'aaseq')
        tofasta.append(kgen)
    f = open(name + '.fasta','w')
    for fasta in tofasta:
        f.write(fasta)
    f.close()
```

8.7.3 Blaster

```
def Blaster(name): #Funciona solo para esta base de datos
    blasteo = NcbiblastpCommandline(query = name + '.fasta', db = '
aa_HST28.fasta', outfmt = 5, out = name + '.xml')
    stdout, stderr = blasteo()
```

8.7.4 HSTRetrieval

```
def HSTRetrieval(gen, resultado):
    e_tres = 0.001
    i_tres = 0.45
    for cadablast in resultado:
        pcode = cadablast.query.split(' ')[0]
```

```
        code = pcode.split('_')[1]
        if code == str(gen):
            print cadablast.query
            print 'RESULTADOS'
            nomark = 0
            reserve = []
            for al in cadablast.alignments:
                for hsp in al.hsps:
                    if hsp.expect <= e_tres:
                        ide = float(hsp.identities)/float(hsp.alig
n_length)
                        if ide >= i_tres:
                            print al.title
                            print 'e: ', hsp.expect
                            print 'identidad: ', ide
                            nomark = nomark+1
                            reserve.append(al.title.split(' ')[1])
            if nomark == 0:
                print 'No se hallaron resultados'
                return None
            elif nomark == 1:
                g = input('Guardar gen? y=1, n=0')
                if g == 0:
                    print 'Gen no guardado'
                    return None
                elif g == 1:
                    print 'Gen guardado'
                    return reserve[0]
            else:
                g = input('Cual guardar? 0 = ninguno')
                if g == 0:
                    print 'Ningún gen guardado'
                    return None
                else:
                    print 'Gen ', str(g), ' guardado'
                    return reserve[g-1]
```

8.7.5 Gene2Rxn

```
def Gene2Rxn(grupo, HST, j, modelo, xlsinfo):
    K = KEGG()
    print 'Obteniendo ortologia de reaccion ',str(j)
    kos = []
    for i in range(len(grupo)):
        if HST[i] != None:
            gen = grupo[i]
            rgen = 'sma:SAVERM_'+ str(gen)
            info = K.get(rgen)
            ortos = K.parse(info)['ORTHOLOGY']
            for key in ortos.keys():
                if key not in kos:
                    kos.append(key)
```

```
    if len(kos) == 1:
        print 'Ortologia identificada: ',kos[0]
        reacID = RxnDown(kos[0], xlsinfo)
    else:
        print 'Hay mas de una ortologia'
        z = 1
        for ko in kos:
            print str(z), ') ', ko
            z = z+1
        reacask = input('ingrese numero')
        reacID = RxnDown(kos[reacask-1], xlsinfo)
    HST2p = []
    for hst in HST:
        if hst != None:
            HST2p.append(hst)
    HST2 = []
    if len(HST2p) > 0:
        HST2.append(HST2p[0])
        for hst2 in HST2p:
            if hst2 not in HST2:
                HST2.append(hst2)
    inistr = '( '
    midstr = ' or '
    finstr = ' )'
    if len(HST2) == 1:
        totstr = inistr+HST2[0]+finstr
    else:
        totstr = inistr
        for i in range(len(HST2)-1):
            totstr = totstr+HST2[i]+midstr
        totstr = totstr+HST2[i+1]+finstr
    AddRxn(modelo, reacID[0], reacID[1], reacID[2], totstr)
    WriteByMe(totstr, xlsinfo)
```

8.7.6 RxnDown

```
def RxnDown(ko, ABM):
    k = KEGG()
    koinfo = k.get(ko)
    rnstr = k.parse(koinfo)['DBLINKS']['RN']
    rnids = rnstr.split(' ')
    if len(rnids) == 1:
        print 'Reaccion encontrada'
    else:
        print 'Se encontraron ',str(len(rnids)), 'reacciones:'
    ind = 1
    for rnid in rnids:
        rn = k.get(rnid)
        rneq = k.parse(rn)['EQUATION']
        print str(ind),') ', rnid, '  ',rneq
        ind = ind +1
```

```
    if len(rnids) == 1:
        actrn = rnids[0]
    else:
        #actid = input('Marque reaccion a utilizar')
        actid = AnswerByMe(ABM[0], ABM[1], 'E', ABM[2], rnids)
        actrn = rnids[actid - 1]
    #revask = input('Es reversible? y=1 n=0')
    revask = AnswerByMe(ABM[0], ABM[1], 'H', ABM[2], None)
    if revask == 1:
        tinfo = [actrn, 1, 1]
    else:
        #dirask = input('Esta en el orden correcto? y=1 n=0')
        ptans = k.get(actrn)
        toans = k.parse(ptans)['EQUATION']
        dirask = AnswerByMe(ABM[0], ABM[1], 'F', ABM[2], toans)
        if dirask == 1:
            tinfo = [actrn, 1, 0]
        else:
            tinfo = [actrn, -1, 0]
    return tinfo
```

8.7.7 AnswerByMe

```
def AnswerByMe(pagina, inirow, task, rxnn, varia):
    fila = inirow + 1 + rxnn
    sfila = str(fila)
    if task == 'E':
        celda = task+sfila
        code = pagina[celda].value
        for i in range(len(varia)):
            if varia[i] == code:
                print 'Se escoge reaccion ' + str(i+1)
                return i+1
    elif task == 'H':
        celda = task+sfila
        yn = pagina[celda].value
        if yn == 'si':
            print 'Reaccion reversible'
            return 1
        elif yn == 'no':
            print 'Reaccion irreversible'
            return 0
    elif task == 'F':
        inic = task+sfila
        finc = 'G' + sfila
        ini = pagina[inic].value
        fin = pagina[finc].value
        inifin = varia.split('<=>')
        ini2 = inifin[0]
        fin2 = inifin[1]
        ini3 = ini2.split(' ')
```

```
        fin3 = fin2.split(' ')
        ans = []
        print ini3
        print fin3
        for meta in ini3:
            if meta == ini:
                ans.append(1)
            elif meta == fin:
                ans.append(-1)
            else:
                print meta + 'falla'
        for meta in fin3:
            if meta == fin:
                ans.append(1)
            elif meta == ini:
                ans.append(-1)
        real = sum(ans)
        if real == 2:
            print 'Reaccion en orden correcto'
            return 1
        elif real == -2:
            print 'Reaccion en orden inverso'
            return 0
        else:
            print 'Hay un problema. Ingresar a mano'
            answer = input('1 si es correcto, 0 si es inverso')
            return answer
```

8.7.8 AddRxn

```
def AddRxn(modelo, rxnID, dire, rev, genstr): # dire = direccion.
1 = de izq a der o reversible. -1 = de der a izq
    k = KEGG()
    if rxnID not in modelo.reactions:
        rxn = k.get(rxnID)
        data = k.parse(rxn)
        if 'NAME' in data.keys():
            nombre = data['NAME']
        else:
            nombre = input('Reaccion sin nombre')
        subs = input('Ingrese subsistema')
        r = Reaction(rxnID)
        r.name = nombre
        r.subsystem = subs
        r.upper_bound = 1000
        if rev == 1:
            r.lower_bound = -1000
        else:
            r.lower_bound = 0
        eq = data['EQUATION']
        sides = eq.split('<=>')
        izq = sides[0]
```

```
22          der = sides[1]
23          clist1 = izq.split(' ')
24          print clist1
25          for i in range(len(clist1)):
26              item = clist1[i]
27              if len(item) >0 and item[0] == 'C':
28                  if i == 0:
29                      est = 1
30                  else:
31                      x = clist1[i-1]
32                      if x.isdigit() == True:
33                          print x
34                          est = int(x)
35                      else:
36                          est = 1
37                  print 'Intentando agregar ',item
38                  meta = AddMet(modelo, item)
39                  coef = -1*est*dire
40                  print coef
41                  r.add_metabolites({meta: coef})
                    print 'El metabolito '+ meta.id + ' (' + meta.name
42  + ') ha sido agregado a la reaccion'
43          clist2 = der.split(' ')
44          print clist2
45          for i in range(len(clist2)):
46              item = clist2[i]
47              if len(item) >0 and item[0] == 'C':
48                  if i == 0:
49                      est = 1
50                  else:
51                      x = clist2[i-1]
52                      if x.isdigit() == True:
53                          print x
54                          est = int(x)
55                      else:
56                          est = 1
57                  print 'Intentando agregar ',item
58                  meta = AddMet(modelo, item)
59                  coef = 1*est*dire
60                  print coef
61                  r.add_metabolites({meta: coef})
                    print 'El metabolito '+ meta.id + ' (' + meta.name
62  + ') ha sido agregado a la reaccion'
63          if len(genstr) > 1:
64              r.gene_reaction_rule = genstr
65          modelo.add_reactions([r])
66          print 'Reaccion agregada'
67      else:
68          print 'La reaccion ya está registrada'
```

8.7.9 AddMet

```
def AddMet(mode, metID):
    K = KEGG()
    cid = metID + '_c'
    if cid not in mode.metabolites:
        met = K.get(metID)
        metdet = K.parse(met)
        if 'FORMULA' not in metdet.keys():
            print 'Formula no encontrada'
            Formula = input('Ingrese formula')
        else:
            Formula = metdet['FORMULA']
        Name = metdet['NAME'][0]
        mmet = Metabolite(cid, formula = Formula, name = Name, com
partment = 'c')
        return mmet
    else:
        print 'El metabolito ya esta registrado'
        return mode.metabolites.get_by_id(cid)
```

8.7.10 WriteByMe

```
def WriteByMe(string, info):
    fila = info[1] + 1 + info[2]
    sfila = str(fila)
    cel = 'I'+sfila
    info[0][cel] = string
```

8.7.11 xlsMaker

```
def xlsMaker(xml, xls, hoja)
    hsma = open(xml)
    notas = load_workbook(xls)
    hoja = notas[hoja]
    sma = Kp.read(hsma)
    node_ids =[]
    index = 0
    for reac in sma.reactions:
        orgcode = reac.entry.name
        ocodes = orgcode.split(' ')
        genstr = '['
        for ocode in ocodes:
            rcode = ocode[11:]
            genstr = genstr + rcode + ', '
        genstr = genstr[0:-2]
        genstr = genstr + '],'
        ide = reac.id
        node_ids.append(ide)
```

```
        name0 = reac.name
        sub = reac.substrates[0].name[4:]
        prod = reac.products[0].name[4:]
        tipo = reac.type
        if tipo == 'reversible':
            rev = 1
        elif tipo == 'irreversible':
            rev = 0
        if len(name0) == 9:
            name = name0[3:]
            defi = GetDef(name)
            index = index + 1
            WriteEverything(hoja, index, name, sub, prod, rev, def
i, genstr)
        else:
            names = name0.split(' ')
            for name1 in names:
                name = name1[3:]
                defi = GetDef(name)
                index = index + 1
                WriteEverything(hoja, index, name, sub, prod, rev,
 defi, genstr)
    notas.save(xls)
```

8.7.12 GetDef

```
def GetDef(code):
    k = KEGG()
    reac = k.get(code)
    defi = k.parse(reac)['DEFINITION']
    return defi
```

8.7.13 WriteEverything

```
def WriteEverything(hoja, index, rxn, s, p, rev, defi, genstr):
    idw = index + 3
    sidw = str(idw)
    hoja['B'+sidw].value = index
    hoja['C'+sidw].value = genstr
    hoja['D'+sidw].value = defi
    hoja['E'+sidw].value = rxn
    hoja['F'+sidw].value = s
    hoja['G'+sidw].value = p
    if rev == 0:
        hoja['H'+sidw].value = 'no'
    elif rev == 1:
        hoja['H'+sidw].value = 'si'
```

8.7.14 EXmaker

```
def EXmaker(modelo, codigo, tipo):
    base = modelo.metabolites.get_by_id(codigo + '_c')
    code = codigo + '_e'
    if code in modelo.metabolites:
        neo = modelo.metabolites.get_by_id(code)
    else:
        neo = Metabolite(codigo + '_e', formula = base.formula, na
me = base.name, compartment = 'e')
    if tipo == 0: # Difusion simple o facilitada
        rname = 't' + codigo + 'd'
        if rname in modelo.reactions:
            print 'Nombre ocupado'
            extra = input('Ingrese identificador')
            rname = rname + extra
        r = Reaction(rname)
        r.name = base.name + ' transport'
        r.lower_bound = -1000
        r.upper_bound = 1000
        modelo.add_reactions([r])
        r.add_metabolites({base: 1,
                           neo: -1})
    if tipo == 1: # Difusion por ABC
        rname = 't' + codigo + 'a'
        if rname in modelo.reactions:
            print 'Nombre ocupado'
            extra = input('Ingrese identificador')
            rname = rname + extra
        r = Reaction(rname)
        r.name = base.name + ' transport via ABC'
        r.lower_bound = 0
        r.upper_bound = 1000
        modelo.add_reactions([r])
        r.add_metabolites({neo: -1,
                           'C00002_c': -1,
                           'C00001_c': -1,
                           base: 1,
                           'C00008_c': 1,
                           'C00009_c': 1})
    if tipo == 2: # Simporte de protones
        rname = 't' + codigo + 'h'
        if rname in modelo.reactions:
            print 'Nombre ocupado'
            extra = input('Ingrese identificador')
            rname = rname + extra
        r = Reaction(rname)
        r.name = base.name + ' transport via proton symport'
        r.upper_bound = 1000
        laask = input('Es reversible? 0 = no, 1 = si')
        r.lower_bound = -1000*laask
        modelo.add_reactions([r])
```

```
        hnum = input('Ingrese numero de protones')
        inout = input('Ingrese sentido, 1 = ingreso o rev, -1 = salida')
        r.add_metabolites({neo: -1*inout,
                           'C00080_e': -hnum*inout,
                           'C00080_c': hnum*inout,
                           base: 1*inout})
    if tipo == 3: # Simporte con sodio
        rname = 't' + codigo + 'n'
        if rname in modelo.reactions:
            print 'Nombre ocupado'
            extra = input('Ingrese identificador')
            rname = rname + extra
        r = Reaction(rname)
        r.name = base.name + ' transport via sodium symport'
        r.upper_bound = 1000
        r.lower_bound = 0
        modelo.add_reactions([r])
        r.add_metabolites({neo: -1,
                           'C01330_e': -1,
                           'C01330_c': 1,
                           base: 1})
    exn = 'EX_' + codigo
    if exn not in modelo.reactions:
        exr = Reaction(exn)
        exr.name = base.name + ' exchange'
        exr.lower_bound = -1000
        exr.upper_bound = 1000
        modelo.add_reactions([exr])
        exr.add_metabolites({neo: -1})
```

8.7.15 errTracker

```
def errTracker(modelo, rxn, xls):
    rxnstd = modelo.reactions.get_by_id(rxn)
    ids = []
    EX = load_workbook(xls)
    Base = EX['Base']
    Raw = EX['Raw']
    for meta in rxnstd.metabolites:
        metaid = meta.id
        toids = metaid[0:-2]
        ex = 'EX_'+toids
        if ex not in modelo.reactions:
            ids.append(toids)
    for ide in ids:
        print ids.index(ide), ide
    for row in Base.rows:
        for cell in row:
            dec = cell.value
            pbin = bin(dec)
            pbina = pbin[2:]
```

```
            if len(pbina) <= len(ids):
                if len(pbina) < len(ids):
                    while len(pbina) < len(ids):
                        pbina = '0' + pbina
                with modelo as modelo:
                    ma = len(ids)
                    lim = ma - 1
                    for i in range(ma):
                        if pbina[i] == '1':
                            actid = ids[lim-i]
                            EXmaker(modelo, actid, 0)
                    modelo.objective = rxnstd
                    opt = modelo.optimize()
                    val = opt.objective_value
                    c = cell.column
                    r = str(cell.row)
                    Raw[c+r].value = val
    EX.save(xls)
```